

Community Experience Distilled

Node.js Design Patterns

Second Edition

Get the best out of Node.js by mastering a series of patterns and techniques to create modular, scalable, and efficient applications

Mario Casciaro

Luciano Mammino

[PACKT] PUBLISHING open source*

目錄

Node.js Design Patterns Second Edition读书笔记

本系列文章为《Node.js Design Patterns Second Edition》的原文翻译和读书笔记，在GitHub连载更新，同步翻译版链接。

欢迎关注我的专栏，之后的博文将在专栏同步：

- Encounter的掘金专栏
- 知乎专栏 Encounter的编程思考
- segmentfault专栏 前端小站

原书PDF下载链接

- 《Node.js Design Patterns Second Edition》

同步翻译目录

> 之后的内容正在阅读中，敬请期待......

投喂

如果您觉得本系列文章对您有所帮助，您可以选择请我吃一包辣条。

```
(function(isHelpful) {
  if (isHelpful) {
    // Spicy dry tofu
    feedToEncounter();
  } else {
    giveEncounterIssue();
  }
})();
```

微信支付	支付宝
推荐使用微信支付 counterxin...(**峰) 微信支付	支付宝 ALIPAY 打开支付宝[扫一扫] Encounter 免费寄送收钱码：拨打95188-6

Welcom to the Node.js Platform

Node.js 的发展

- 技术本身的发展
- 庞大的 `Node.js` 生态圈的发展
- 官方组织的维护

Node.js的特点

小模块

以 `package` 的形式尽可能多的复用模块，原则上每个模块的容量尽量小而精。

原则：

- "Small is beautiful" ---小而精
- "Make each program do one thing well" ---单一职责原则

因此，一个 `Node.js` 应用由多个包搭建而成，包管理器（ `npm` ）的管理使得他们相互依赖而不起冲突。

如果设计一个 `Node.js` 的模块，尽可能做到以下三点：

- 易于理解和使用
- 易于测试和维护
- 考虑到对客户端（浏览器）的支持更友好

以及， `Don't Repeat Yourself(DRY)` 复用性原则。

以接口形式提供

每个 `Node.js` 模块都是一个函数（类也是以构造函数的形式呈现），我们只需要调用相关 `API` 即可，而不需要知道其它模块的实现。 `Node.js` 模块是为了使用它们而创建，不仅仅是在拓展性上，更要考虑到维护性和可用性。

简单且实用

> “简单就是终极的复杂”————达尔文

遵循 `KISS(Keep It Simple, Stupid)原则` ，即优秀的简洁的设计，能够更有效地传递信息。

设计必须很简单，无论在实现还是接口上，更重要的是实现比接口更简单，简单是重要的设计原则。

我们做一个设计简单，功能完备，而不是完美的软件：

- 实现起来需要更少的努力
- 允许用更少的速度进行更快的运输资源
- 具有伸缩性，更易于维护和理解
- 促进社区贡献，允许软件本身的成长和改进

而对于 `Node.js` 而言，因为其支持 `JavaScript` ，简单和函数、闭包、对象等特性，可取代复杂的面向对象的类语法。如单例模式和装饰者模式，它们在面向对象的语言都需要很复杂的实现，而对于 `JavaScript` 则较为简单。

介绍Node.js 6 和 ES2015的新语法

let和const关键字

`ES5` 之前，只有函数和全局作用域。

```
if (false) {
  var x = "hello";
}

console.log(x); // undefined
```

现在用 `let` ，创建词法作用域，则会报出一个错误 `Uncaught ReferenceError: x is not defined`

```
if (false) {
  let x = "hello";
}

console.log(x);
```

在循环语句中使用 `let` ，也会报错 `Uncaught ReferenceError: i is not defined` ：

```
for (let i = 0; i < 10; i++) {
  // do something here
}

console.log(i);
```

使用 `let` 和 `const` 关键字，可以让代码更安全，如果意外的访问另一个作用域的变量，更容易发现错误。

使用 `const` 关键字声明变量，变量不会被意外更改。

```
const x = 'This will never change';
x = '...';
```

这里会报出一个错误 `Uncaught TypeError: Assignment to constant variable.`

但是对于对象属性的更改，`const` 显得毫无办法：

```
const x = {};
x.name = 'John';
```

上述代码并不会报错

但是如果直接更改对象，还是会抛出一个错误。

```
const x = {};
x = null;
```

实际运用中，我们使用 `const` 引入模块，防止意外被更改：

```
const path = require('path');
let path = './some/path';
```

上述代码会报错，提醒我们意外更改了模块。

如果需要创建不可变对象，只是简单的使用 `const` 是不够的，需要使用 `Object.freeze()` 或deep-freeze

我看了一下源码，其实很少，就是递归使用 `Object.freeze()`

```
module.exports = function deepFreeze (o) {
  Object.freeze(o);

  Object.getOwnPropertyNames(o).forEach(function (prop) {
    if (o.hasOwnProperty(prop)
    && o[prop] !== null
    && (typeof o[prop] === "object" || typeof o[prop] === "funct
ion")
    && !Object.isFrozen(o[prop])) {
      deepFreeze(o[prop]);
    }
  });

  return o;
};
```

箭头函数

箭头函数更易于理解，特别是在我们定义回调的时候：

```
const numbers = [2, 6, 7, 8, 1];
const even = numbers.filter(function(x) {
  return x % 2 === 0;
});
```

使用箭头函数语法，更简洁：

```
const numbers = [2, 6, 7, 8, 1];
const even = numbers.filter(x => x % 2 === 0);
```

如果不止一个 `return` 语句则使用 `=> {}`

```
const numbers = [2, 6, 7, 8, 1];
const even = numbers.filter((x) => {
  if (x % 2 === 0) {
    console.log(x + ' is even');
    return true;
  }
});
```

最重要是，箭头函数绑定了它的词法作用域，其 `this` 与父级代码块的 `this` 相同。

```
function DelayedGreeter(name) {
  this.name = name;
}

DelayedGreeter.prototype.greet = function() {
  setTimeout(function cb() {
    console.log('Hello' + this.name);
  }, 500);
}

const greeter = new DelayedGreeter('World');
greeter.greet(); // 'Hello'
```

要解决这个问题，使用箭头函数或 `bind`

```
function DelayedGreeter(name) {
  this.name = name;
}

DelayedGreeter.prototype.greet = function() {
  setTimeout(function cb() {
    console.log('Hello' + this.name);
  }.bind(this), 500);
}

const greeter = new DelayedGreeter('World');
greeter.greet(); // 'HelloWorld'
```

或者箭头函数，与父级代码块作用域相同：

```
function DelayedGreeter(name) {
  this.name = name;
}

DelayedGreeter.prototype.greet = function() {
  setTimeout(() => console.log('Hello' + this.name), 500);
}

const greeter = new DelayedGreeter('World');
greeter.greet(); // 'HelloWorld'
```

类语法糖

`class` 是原型继承的语法糖，对于来自传统的面向对象语言的所有开发人员（如 `Java` 和 `C#` ）来说更熟悉，新语法并没有改变 `JavaScript` 的运行特征，通过原型来完成更加方便和易读。

传统的通过`构造器 + 原型`的写法：

```
function Person(name, surname, age) {
  this.name = name;
  this.surname = surname;
  this.age = age;
}

Person.prototype.getFullName = function() {
  return this.name + '' + this.surname;
}

Person.older = function(person1, person2) {
  return (person1.age >= person2.age) ? person1 : person2;
}
```

使用`class`语法显得更加简洁、方便、易懂：

```
class Person {
  constructor(name, surname, age) {
    this.name = name;
    this.surname = surname;
    this.age = age;
  }

  getFullName() {
    return this.name + '' + this.surname;
  }

  static older(person1, person2) {
    return (person1.age >= person2.age) ? person1 : person2;
  }
}
```

但是上面的实现是可以互换的，但是，对于`class`语法来说，最有意义的是`extends`和`super`关键字。

```
class PersonWithMiddlename extends Person {
  constructor(name, middlename, surname, age) {
    super(name, surname, age);
    this.middlename = middlename;
  }

  getFullName() {
    return this.name + '' + this.middlename + '' + this.surname;
  }
}
```

这个例子是真正的面向对象的方式，我们声明了一个希望被继承的类，定义新的构造器，并可以使用 `super` 关键字调用父构造器，并重写 `getFullName` 方法，使得其支持 `middlename` 。

对象字面量的新语法

允许缺省值：

```
const x = 22;
const y = 17;
const obj = { x, y };
```

允许省略方法名

```
module.exports = {
  square(x) {
    return x * x;
  },
  cube(x) {
    return x * x * x;
  },
};
```

key的计算属性

```
const namespace = '-webkit-';
const style = {
  [namespace + 'box-sizing']: 'border-box',
  [namespace + 'box-shadow']: '10px 10px 5px #888',
};
```

新的定义getter和setter方式

```
const person = {
  name: 'George',
  surname: 'Boole',

  get fullname() {
    return this.name + ' ' + this.surname;
  },

  set fullname(fullname) {
    let parts = fullname.split(' ');
    this.name = parts[0];
    this.surname = parts[1];
  }
};

console.log(person.fullname); // "George Boole"
console.log(person.fullname = 'Alan Turing'); // "Alan Turing"
console.log(person.name); // "Alan"
```

这里，第二个 `console.log` 触发了 `set` 方法。

模板字符串

其它ES2015语法

- 函数默认参数
- 剩余参数语法
- 拓展运算符
- 解构赋值
- new.target
- 代理
- 反射
- Symbol

reactor模式

`reactor模式` 是 `Node.js` 异步编程的核心模块，其核心概念是： `单线程` 、 `非阻塞I/O` ，通过下列例子可以看到 `reactor模式` 在 `Node.js` 平台的体现。

I/O是缓慢的

在计算机的基本操作中，输入输出肯定是最慢的。访问内存的速度是纳秒级(`10e-9 s`)，同时访问磁盘上的数据或访问网络上的数据则更慢，是毫秒级(`10e-3 s`)。内存的传输速度一般认为是 `GB/s` 来计算，然而磁盘或网络的访问

速度则比较慢，一般是 `MB/s` 。虽然对于 `CPU` 而言， `I/O` 操作的资源消耗并不算大，但是在发送 `I/O` 请求和操作完成之间总会存在时间延迟。除此之外，我们还必须考虑人为因素，通常情况下，应用程序的输入是人为产生的，例如：按钮的点击、即时聊天工具的信息发送。因此，输入输出的速度并不因网络和磁盘访问速率慢造成的，还有多方面的因素。

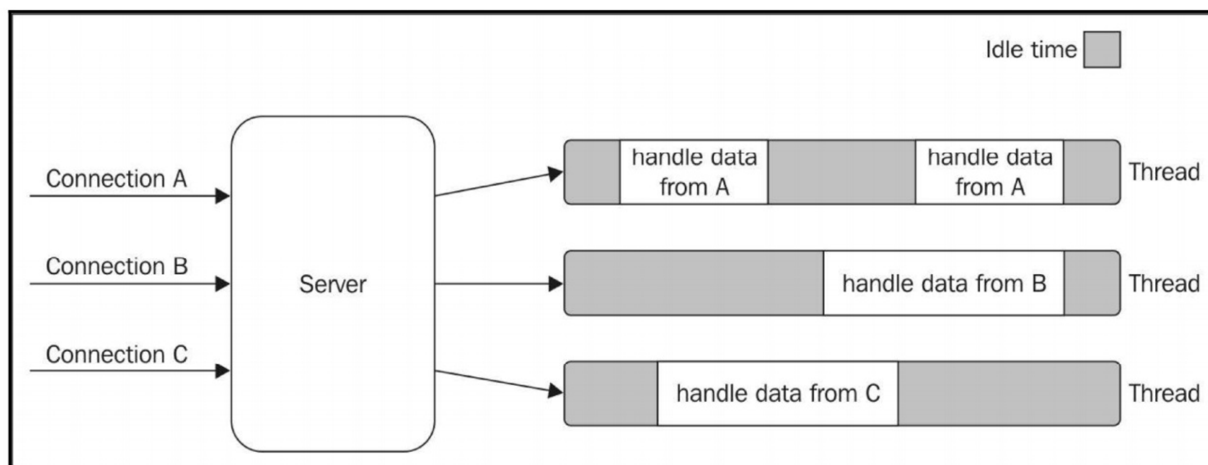
阻塞I/O

在一个 `阻塞I/O` 模型的进程中， `I/O` 请求会阻塞之后代码块的运行。在 `I/O` 请求操作完成之前，线程会有一段不定长的时间浪费。（它可能是毫秒级的，但甚至有可能是分钟级的，如用户按着一个按键不放的情况）。以下例子就是一个 `阻塞I/O` 模型。

```
// 直到请求完成，数据可用，线程都是阻塞的
data = socket.read();
// 请求完成，数据可用
print(data);
```

我们知道， `阻塞I/O` 的服务器模型并不能在一个线程中处理多个连接，每次 `I/O` 都会阻塞其它连接的处理。出于这个原因，对于每个需要处理的并发连接，传统的web服务器的处理方式是新开一个新的进程或线程（或者从线程池中重用一个进程）。这样，当一个线程因 `I/O` 操作被阻塞时，它并不会影响另一个线程的可用性，因为他们是在彼此独立的线程中处理的。

通过下面这张图：



通过上面的图片我们可以看到每个线程都有一段时间处于空闲等待状态，等待从关联连接接收新数据。如果所有种类的 `I/O` 操作都会阻塞后续请求。例如，连接数据库和访问文件系统，现在我们能很快知晓一个线程需要因等待 `I/O` 操作的结果等待许多时间。不幸的是，一个线程所持有的 `CPU` 资源并不廉价，它需要消耗内存、造成 `CPU` 上下文切换，因此，长期占有 `CPU` 而大部分时间并没有使用的线程，在资源利用率上考虑，并不是高效的选择。

非阻塞I/O

除 `阻塞I/O` 之外，大部分现代的操作系统支持另外一种访问资源的机制，即 `非阻塞I/O` 。在这种机制下，后续代码块不会等到 `I/O` 请求数据的返回之后再执行。如果当前时刻所有数据都不可用，函数会先返回预先定义的常量值(如 `undefined`)，表明当前时刻暂无数据可用。

例如，在 `Unix` 操作系统中， `fcntl()` 函数操作一个已存在的文件描述符，改变其操作模式为 `非阻塞I/O` (通过 `O_NONBLOCK` 状态字)。一旦资源是非阻塞模式，如果读取文件操作没有可读取的数据,或者如果写文件操作被阻塞,读操作或写操作返回 `-1` 和 `EAGAIN` 错误。

`非阻塞I/O` 最基本的模式是通过轮询获取数据，这也叫做忙-等模型。看下面这个例子，通过 `非阻塞I/O` 和轮询机制获取 `I/O` 的结果。

```
resources = [socketA, socketB, pipeA];
while(!resources.isEmpty()) {
  for (i = 0; i < resources.length; i++) {
    resource = resources[i];
    // 进行读操作
    let data = resource.read();
    if (data === NO_DATA_AVAILABLE) {
      // 此时还没有数据
      continue;
    }
    if (data === RESOURCE_CLOSED) {
      // 资源被释放，从队列中移除该链接
      resources.remove(i);
    } else {
      consumeData(data);
    }
  }
}
```

我们可以看到，通过这个简单的技术，已经可以在一个线程中处理不同的资源了，但依然不是高效的。事实上，在前面的例子中，用于迭代资源的循环只会消耗宝贵的 `CPU` ，而这些资源的浪费比起 `阻塞I/O` 反而更不可接受，轮询算法通常浪费大量 `CPU` 时间。

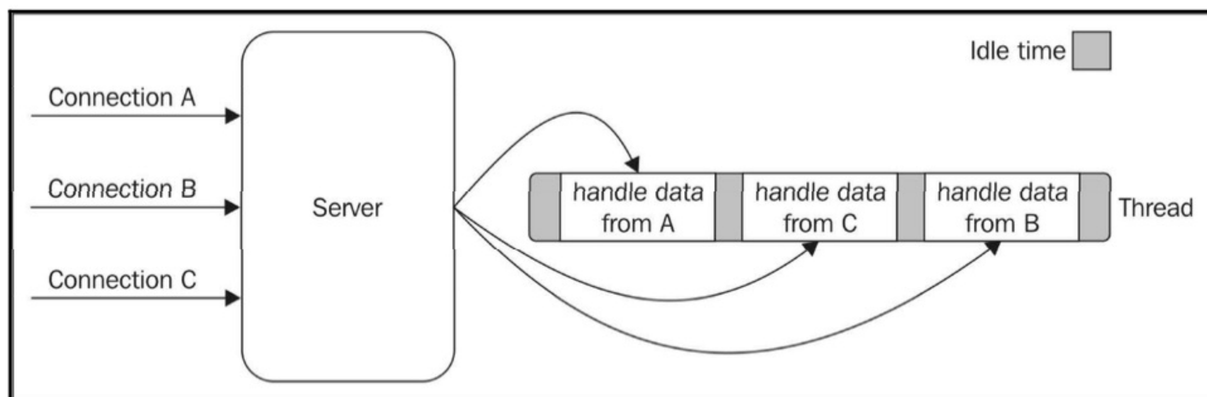
事件多路复用

对于获取非阻塞的资源而言， `忙-等模型` 不是一个理想的技术。但是幸运的是，大多数现代的操作系统提供了一个原生的机制来处理并发，非阻塞资源（同步事件多路复用器）是一个有效的方法。这种机制被称作事件循环机制，这种事件收集和 `I/O队列` 源于 `发布-订阅模式` 。事件多路复用器收集资源的 `I/O` 事件并且把这些事件放入队列中，直到事件被处理时都是阻塞状态。看下面这个伪代码：

```
socketA, pipeB;
wachedList.add(socketA, FOR_READ);
wachedList.add(pipeB, FOR_READ);
while(events = demultiplexer.watch(wachedList)) {
  // 事件循环
  foreach(event in events) {
    // 这里并不会阻塞，并且总会有返回值（不管是不是确切的值）
    data = event.resource.read();
    if (data === RESOURCE_CLOSED) {
      // 资源已经被释放，从观察者队列移除
      demultiplexer.unwatch(event.resource);
    } else {
      // 成功拿到资源，放入缓冲池
      consumeData(data);
    }
  }
}
```

事件多路复用的三个步骤：

- 资源被添加到一个数据结构中，为每个资源关联一个特定的操作，在这个例子中是 `read` 。
- 事件通知器由一组被观察的资源组成，一旦事件即将触发，会调用同步的 `watch` 函数，并返回这个可被处理的事件。
- 最后，处理事件多路复用器返回的每个事件，此时，与系统资源相关联的事件将被读并且在整个操作中都是非阻塞的。直到所有事件都被处理完时，事件多路复用器会再次阻塞，然后重复这个步骤，以上就是 `event loop` 。



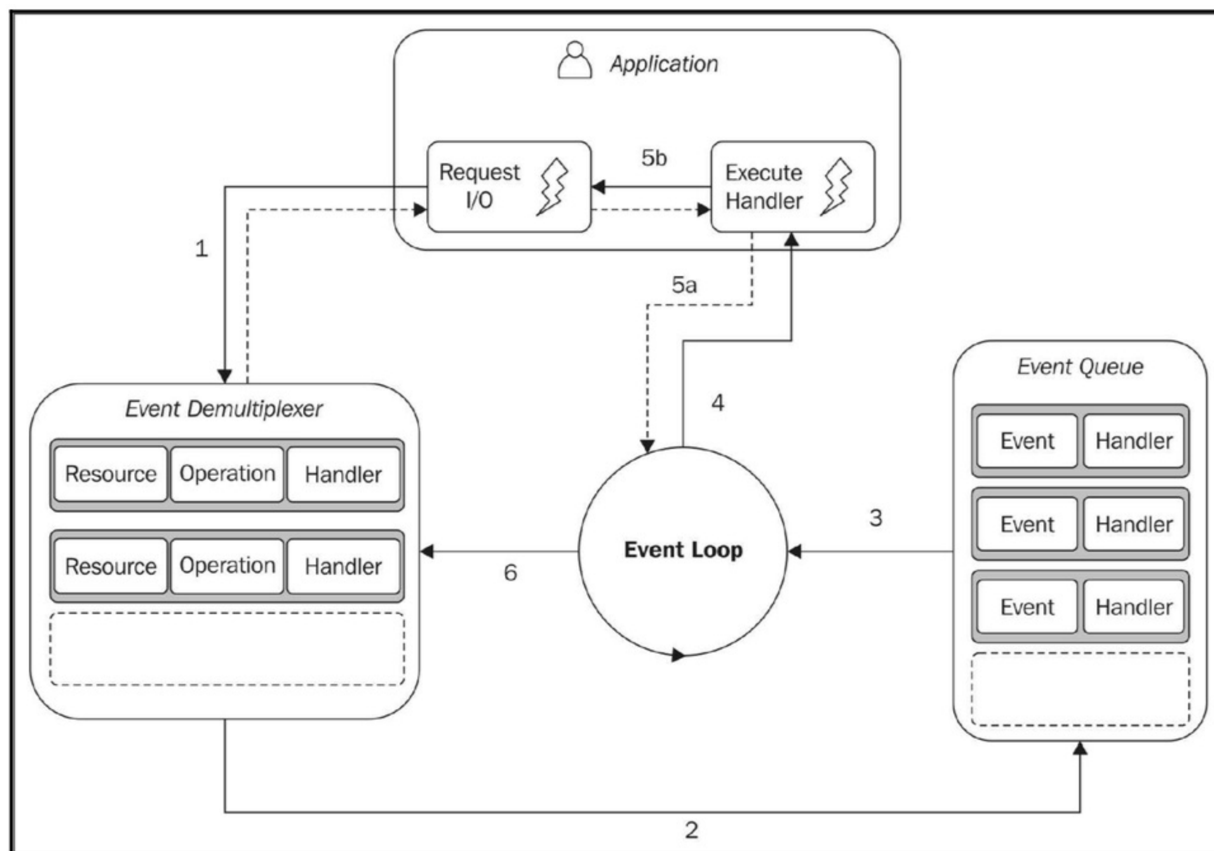
上图可以很好的帮助我们理解在一个单线程的应用程序中使用同步的时间多路复用器和非阻塞 `I/O` 实现并发。我们能够看到，只使用一个线程并不会影响我们处理多个 `I/O` 任务的性能。同时，我们看到任务是在单个线程中随着时间的推移而展开的，而不是分散在多个线程中。我们看到，在单线程中传播的任务相对于多线程中传播的任务反而节约了线程的总体空闲时间，并且更利于程序员编写代码。在这本书中，你可以看到我们可以用更简单的并发策略，因为不需要考虑多线程的互斥和同步问题。

在下一章中，我们有更多机会讨论 `Node.js` 的并发模型。

介绍reactor模式

现在来说 `reactor模式` ，它通过一种特殊的算法设计的处理程序（在 `Node.js` 中是使用一个回调函数表示），一旦事件产生并在事件循环中被处理，那么相关 `handler` 将会被调用。

它的结构如图所示：



`reactor模式` 的步骤为：

- 应用程序通过提交请求到时间多路复用器产生一个新的 `I/O` 操作。应用程序指定 `handler` ， `handler` 在操作完成后被调用。提交请求到事件多路复用器是非阻塞的，其调用所以会立马返回，将执行权返回给应用程序。
- 当一组 `I/O` 操作完成，事件多路复用器会将这些新事件添加到事件循环队列中。
- 此时，事件循环会迭代事件循环队列中的每个事件。
- 对于每个事件，对应的 `handler` 被处理。
- `handler` ，是应用程序代码的一部分， `handler` 执行结束后执行权会交回事件循环。但是，在 `handler` 执行时可能请求新的异步操作，从而新的操作被添加到事件多路复用器。
- 当事件循环队列的全部事件被处理完后，循环会在事件多路复用器再次阻塞直到有一个新的事件可处理触发下一次循环。

我们现在可以定义 `Node.js` 的核心模式：

模式(反应器)阻塞处理 `I/O` 到在一组观察的资源有新的事件可处理，然后以分派每个事件对应 `handler` 的方式反应。

OS的非阻塞I/O引擎

每个操作系统对于事件多路复用器有其自身的接口，`Linux` 是 `epoll` ，`Mac OSX` 是 `kqueue` ，`Windows` 的 `IOCP API` 。除外，即使在相同的操作系统中，每个 `I/O` 操作对于不同的资源表现不一样。例如，在 `Unix` 下，普通文件系统不支持非阻塞操作，所以，为了模拟非阻塞行为，需要使用在事件循环外用一个独立的线程。所有这些平台内和跨平台的不一致性需要在事件多路复用器的上层做抽象。这就是为什么 `Node.js` 为了兼容所有主流平台而 编写C语言库 `libuv` ，目的就是为了使得 `Node.js` 兼容所有主流平台和规范化不同类型资源的非阻塞行为。 `libuv` 今天作为 `Node.js` 的 `I/O` 引擎的底层。

Node.js Essential Patterns

对于 `Node.js` 而言，异步特性是其最显著的特征，但对于别的一些语言，例如 `PHP` ，就不常处理异步代码。

在同步的编程中，我们习惯于把代码的执行想象为自上而下连续的执行计算步骤。每个操作都是阻塞的，这意味着只有在一个操作执行完成后才能执行下一个操作，这种方式利于我们理解和调试。

然而，在异步的编程中，我们可以在后台执行诸如读取文件或执行网络请求的一些操作。当我们在调用异步操作方法时，即使当前或之前的操作尚未完成，下面的后续操作也会继续执行，在后台执行的操作会在任意时刻执行完毕，并且应用程序会在异步调用完成时以正确的方式做出反应。

虽然这种非阻塞方法相比于阻塞方法性能更好，但它实在是让程序员难以理解，并且，在处理较为复杂的异步控制流的高级应用程序时，异步顺序可能会变得难以操作。

`Node.js` 提供了一系列工具和设计模式，以便我们最佳地处理异步代码。了解如何使用它们编写性能和易于理解和调试的应用程序非常重要。

在本章中，我们将看到两个最重要的异步模式：回调和事件发布器。

回调模式

在上一章中介绍过，回调是 `reactor模式` 的 `handler` 的实例，回调本来就是 `Node.js` 独特的编程风格之一。回调函数是在异步操作完成后传播其操作结果的函数，总是用来替代同步操作的返回指令。而 `JavaScript` 恰好就是表示回调的最好的语言。在 `JavaScript` 中，函数是一等公民，我们可以把函数变量作为参数传递，并在另一个函数中调用它，把调用的结果存储到某一数据结构中。实现回调的另一个理想结构是闭包。使用闭包，我们能够保留函数创建时所在的上下文环境，这样，无论何时调用回调，都保持了请求异步操作的上下文。

在本节中，我们分析基于回调的编程思想和模式，而不是同步操作的返回指令的模式。

CPS

在 `JavaScript` 中，回调函数作为参数传递给另一个函数，并在操作完成时调用。在函数式编程中，这种传递结果的方法被称为 `CPS` 。这是一个一般概念，而且不只是对于异步操作而言。实际上，它只是通过将结果作为参数传递给另一个函数（回调函数）来传递结果，然后在主体逻辑中调用回调函数拿到操作结果，而不是直接将其返回给调用者。

同步CPS

为了更清晰地理解 `CPS` ，让我们来看看这个简单的同步函数：

```
function add(a, b) {
  return a + b;
}
```

上面的例子成为直接编程风格，其实没什么特别的，就是使用 `return` 语句把结果直接传递给调用者。它代表的是同步编程中返回结果的最常见方法。上述功能的 `CPS` 写法如下：

```
function add(a, b, callback) {
  callback(a + b);
}
```

`add()` 函数是一个同步的 `CPS` 函数， `CPS` 函数只会在它调用的时候才会拿到 `add()` 函数的执行结果，下列代码就是其调用方式：

```
console.log('before');
add(1, 2, result => console.log('Result: ' + result));
console.log('after');
```

既然 `add()` 是同步的，那么上述代码会打印以下结果：

```
before
Result: 3
after
```

异步CPS

那我们思考下面的这个例子，这里的 `add()` 函数是异步的：

```
function additionAsync(a, b, callback) {
 setTimeout(() => callback(a + b), 100);
}
```

在上边的代码中，我们使用 `setTimeout()` 模拟异步回调函数的调用。现在，我们调用 `additionalAsync` ，并查看具体的输出结果。

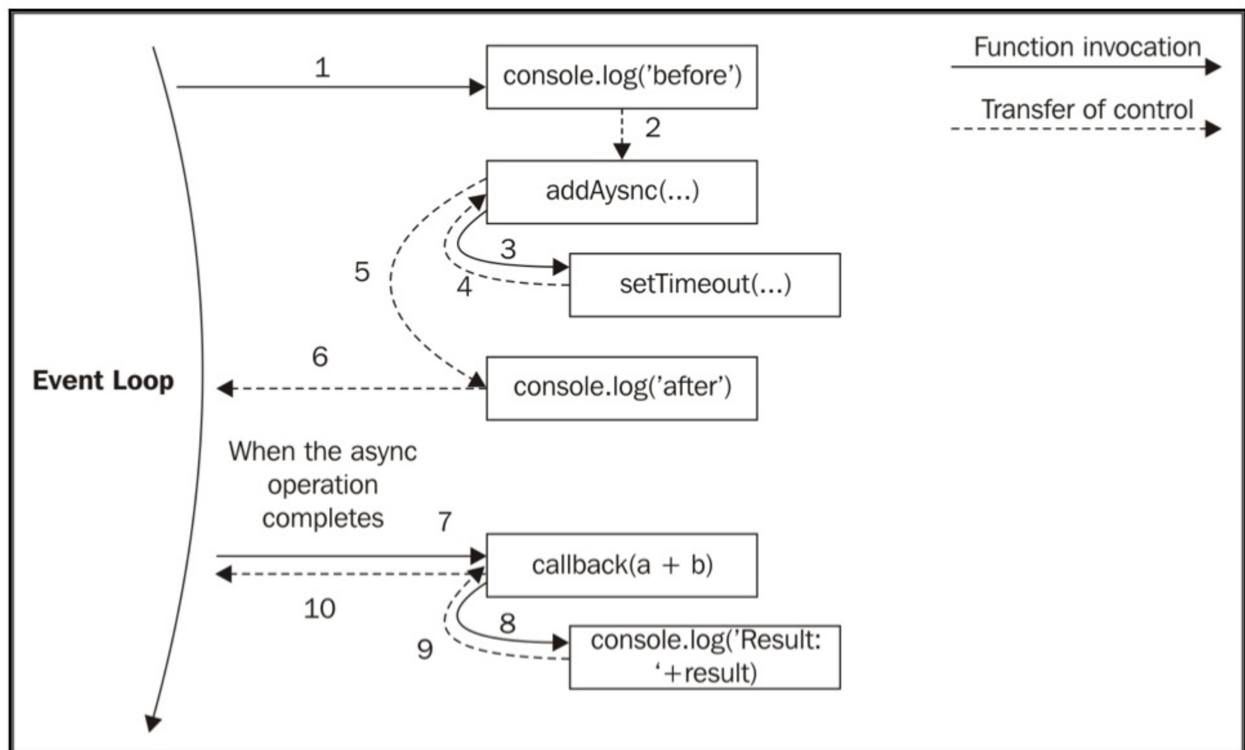
```
console.log('before');
additionAsync(1, 2, result => console.log('Result: ' + result));
console.log('after');
```

上述代码会有以下的输出结果：

```
before
after
Result: 3
```

因为 `setTimeout()` 是一个异步操作，所以它不会等待执行回调，而是立即返回，将控制权交给 `addAsync()` ，然后返回给其调用者。 `Node.js` 中的此属性至关重要，因为只要有异步请求产生，控制权就会交给事件循环，从而允许处理来自队列的新事件。

下面的图片显示了 `Node.js` 中事件循环过程：



当异步操作完成时，执行权就会交给这个异步操作开始的地方，即回调函数。执行将从事件循环开始，所以它将有一个新的堆栈。对于 `JavaScript` 而言，这是它的优势所在。正是由于闭包保存了其上下文环境，即使在不同的时间点和不同的位置调用回调，也能够正常地执行。

同步函数在其完成操作之前是阻塞的。而异步函数立即返回，结果将在事件循环的稍后循环中传递给处理程序（在我们的例子中是一个回调）。

非CPS风格的回调模式

某些情况下情况下，我们可能会认为回调CPS式的写法像是异步的，然而并不是。比如以下代码， `Array` 对象的 `map()` 方法：

```
const result = [1, 5, 7].map(element => element - 1);
console.log(result); // [0, 4, 6]
```

在上述例子中，回调仅用于迭代数组的元素，而不是传递操作的结果。实际上，这个例子中是使用回调的方式同步返回，而非传递结果。是否是传递操作结果的回调通常在 `API` 文档有明确说明。

同步还是异步？

我们已经看到代码的执行顺序会因同步或异步的执行方式产生根本性的改变。这对整个应用程序的流程，正确性和效率都产生了重大影响。以下是对这两种模式及其缺陷的分析。一般来说，必须避免的是由于其执行顺序不一致导致的难以检测和拓展的混乱。下面是一个有陷阱的异步实例：

一个有问题的函数

最危险的情况之一是在特定条件下同步执行本应异步执行的 `API` 。以下列代码为例：

```
const fs = require('fs');
const cache = {};

function inconsistentRead(filename, callback) {
  if (cache[filename]) {
    // 如果缓存命中，则同步执行回调
    callback(cache[filename]);
  } else {
    // 未命中，则执行异步非阻塞的I/O操作
    fs.readFile(filename, 'utf8', (err, data) => {
      cache[filename] = data;
      callback(data);
    });
  }
}
```

上述功能使用缓存来存储不同文件读取操作的结果。不过记得，这只是一个例子，它缺少错误处理，并且其缓存逻辑本身不是最佳的（比如没有缓存淘汰策略）。除此之外，上述函数是非常危险的，因为如果没有设置高速缓存，它的行为是异步的，直到 `fs.readFile()` 函数返回结果为止，它都不会同步执行，这时缓存并不会触发，而会去走异步回调调用。

解放zalgo

关于 `zalgo` ，其实就是指同步或异步行为的不确定性，几乎总是导致非常难追踪的 `bug` 。

现在，我们来看看如何使用一个不可预测其顺序的函数，它甚至可以轻松地中断一个应用程序。看以下代码：

```
function createFileReader(filename) {
  const listeners = [];
  inconsistentRead(filename, value => {
    listeners.forEach(listener => listener(value));
  });
  return {
    onDataReady: listener => listeners.push(listener)
  };
}
```

当上述函数被调用时，它创建一个充当事件发布器的新对象，允许我们为文件读取操作设置多个事件监听器。当读取操作完成并且数据可用时，所有的监听器将被立即被调用。前面的函数使用之前定义的 `inconsistentRead()` 函数来实现这个功能。我们现在尝试调用 `createFileReader()` 函数：

```
const reader1 = createFileReader('data.txt');
reader1.onDataReady(data => {
 console.log('First call data: ' + data);
 // 之后再次通过fs读取同一个文件
 const reader2 = createFileReader('data.txt');
 reader2.onDataReady(data => {
   console.log('Second call data: ' + data);
 });
});
```

之后的输出是这样的：

```
➜ 04_callback_unpredictable git:(master) node test.js
First call data: This example shows unpredictable sync/async callbacks.

To run the example launch:

  node test
```

```
First call data: some data
```

下面来分析为何第二次的回调没有被调用：

在创建 `reader1` 的时候，`inconsistentRead()` 函数是异步执行的，这时没有可用的缓存结果，因此我们有时间注册事件监听器。在读操作完成后，它将在下一次事件循环中被调用。

然后，在事件循环的循环中创建 `reader2` ，其中所请求文件的缓存已经存在。在这种情况下，内部调用 `inconsistentRead()` 将是同步的。所以，它的回调将被立即调用，这意味着 `reader2` 的所有监听器也将被同步调用。然而，在创建 `reader2` 之后，我们才开始注册监听器，所以它们将永远不被调用。

`inconsistentRead()` 回调函数的行为是不可预测的，因为它取决于许多因素，例如调用的频率，作为参数传递的文件名，以及加载文件所花费的时间等。

在实际应用中，例如我们刚刚看到的错误可能会非常复杂，难以在真实应用程序中识别和复制。想象一下，在 `Web服务器` 中使用类似的功能，可以有多个并发请求;想象一下这些请求挂起，没有任何明显的理由，没有任何日志被记录。这绝对属于烦人的 `bug` 。

`npm` 的创始人和以前的 `Node.js` 项目负责人 `Isaac Z. Schlueter` 在他的一篇博客文章中比较了使用这种不可预测的功能来释放 `Zalgo` 。如果您不熟悉 `Zalgo` 。可以看看Isaac Z. Schlueter的原始帖子。

使用同步API

从上述关于 `zalgo` 的示例中，我们知道， `API` 必须清楚地定义其性质：是同步的还是异步的？

我们合适 `fix` 上述的 `inconsistentRead()` 函数产生的 `bug` 的方式是使它完全同步阻塞执行。并且这是完全可能的，因为 `Node.js` 为大多数基本 `I/O` 操作提供了一组同步方式的 `API` 。例如，我们可以使用 `fs.readFileSync()` 函数来代替它的异步对等体。代码现在如下：

```
const fs = require('fs');
const cache = {};

function consistentReadSync(filename) {
  if (cache[filename]) {
    return cache[filename];
  } else {
    cache[filename] = fs.readFileSync(filename, 'utf8');
    return cache[filename];
  }
}
```

我们可以看到整个函数被转化为同步阻塞调用的模式。如果一个函数是同步的，那么它不会是 `CPS` 的风格。事实上，我们可以说，使用 `CPS` 来实现一个同步的 `API` 一直是最佳实践，这将消除其性质上的任何混乱，并且从性能角度来看也将更加有效。

请记住，将 `API` 从 `CPS` 更改为直接调用返回的风格，或者说从异步到同步的风格。例如，在我们的例子中，我们必须完全改变我们的 `createFileReader()` 为同步，并使其适应于始终工作。

另外，使用同步 `API` 而不是异步 `API` ，要特别注意以下注意事项：

- 同步 `API` 并不适用于所有应用场景。
- 同步 `API` 将阻塞事件循环并将并发请求置于阻塞状态。它会破坏 `JavaScript` 的并发模型，甚至使得整个应用程序的性能下降。我们将在本书后面看到这对我们的应用程序的影响。

在我们的 `inconsistentRead()` 函数中，因为每个文件名仅调用一次，所以同步阻塞调用而对应用程序造成的影响并不大，并且缓存值将用于所有后续的调用。如果我们的静态文件的数量是有限的，那么使用 `consistentReadSync()` 将不会对我们的事件循环产生很大的影响。如果我们文件数量很大并且都需要被读取一次，而且对性能要求较高的情况下，我们不建议在 `Node.js` 中使用同步 `I/O` 。然而，在某些情况下，同步 `I/O` 可能是最简单和最有效的解决方案。所以我们必须正确评估具体的应用场景，以选择最为合适的方案。上述实例其实说明：在实际应用程序中使用同步阻塞 `API` 加载配置文件是非常有意义的。

因此，记得只有不影响应用程序并发能力时才考虑使用同步阻塞 `I/O` 。

延时处理

另一种 `fix` 上述的 `inconsistentRead()` 函数产生的 `bug` 的方式是让它仅仅是异步的。这里的解决办法是下一次事件循环时同步调用，而不是在相同的事件循环周期中立即运行，使得其实际上是异步的。在 `Node.js` 中，可以使用 `process.nextTick()` ，它延迟函数的执行，直到下一次传递事件循环。它的功能非常简单，它将回调作为参数，并将其推送到事件队列的顶部，在任何未处理的 `I/O` 事件前，并立即返回。一旦事件循环再次运行，就会立刻调用回调。

所以看下列代码，我们可以较好的利用这项技术处理 `inconsistentRead()` 的异步顺序：

```
const fs = require('fs');
const cache = {};

function consistentReadAsync(filename, callback) {
  if (cache[filename]) {
    // 下一次事件循环立即调用
    process.nextTick(() => callback(cache[filename]));
  } else {
    // 异步I/O操作
    fs.readFile(filename, 'utf8', (err, data) => {
      cache[filename] = data;
      callback(data);
    });
  }
}
```

现在，上述函数保证在任何情况下异步地调用其回调函数，解决了上述 `bug` 。

另一个用于延迟执行代码的 `API` 是 `setImmediate()` 。虽然它们的作用看起来非常相似，但实际含义却截然不同。 `process.nextTick()` 的回调函数会在任何其他 `I/O` 操作之前调用，而对于 `setImmediate()` 则会在其它 `I/O` 操作之后调用。由于 `process.nextTick()` 在其它的 `I/O` 之前调用，因此在某些情况下可能会导致 `I/O` 进入无限期等待，例如递归调用 `process.nextTick()` 但是对于 `setImmediate()` 则不会发生这种情况。当我们在本书后面分析使用延迟调用来运行同步 `CPU` 绑定任务时，我们将深入了解这两种API之间的区别。

我们保证通过使用 `process.nextTick()` 异步调用其回调函数。

Node.js回调风格

对于 `Node.js` 而言， `CPS` 风格的 `API` 和回调函数遵循一组特殊的约定。这些约定不只是适用于 `Node.js` 核心 `API` ，对于它们之后也是绝大多数用户级模块和应用程序也很有意义。因此，我们了解这些风格，并确保我们在需要设计异步 `API` 时遵守规定显得至关重要。

回调总是最后一个参数

在所有核心 `Node.js` 方法中，标准约定是当函数在输入中接受回调时，必须作为最后一个参数传递。我们以下面的 `Node.js` 核心 `API` 为例：

```
fs.readFile(filename, [options], callback);
```

从前面的例子可以看出，即使是在可选参数存在的情况下，回调也始终置于最后的位置。其原因是在回调定义的情况下，函数调用更可读。

错误处理总在最前

在 `CPS` 中，错误以不同于正确结果的形式在回调函数中传递。在 `Node.js` 中， `CPS` 风格的回调函数产生的任何错误总是作为回调的第一个参数传递，并且任何实际的结果从第二个参数开始传递。如果操作成功，没有错误，第一个参数将为 `null` 或 `undefined` 。看下列代码：

```
fs.readFile('foo.txt', 'utf8', (err, data) => {
  if (err)
    handleError(err);
  else
    processData(data);
});
```

上面的例子是最好的检测错误的方法，如果不检测错误，我们可能难以发现和调试代码中的 `bug` ，但另外一个要考虑的问题是错误总是为 `Error` 类型，这意味着简单的字符串或数字不应该作为错误对象传递（难以被 `try catch` 代码块捕

获）。

错误传播

对于同步阻塞的写法而言，我们的错误都是通过 `throw` 语句抛出，即使错误在错误栈中跳转，我们也能很好地捕获到错误上下文。

但是对于 `CPS` 风格的异步调用而言，通过把错误传递到错误栈中的下一个回调来完成，下面是一个典型的例子：

```
const fs = require('fs');

function readJSON(filename, callback) {
  fs.readFile(filename, 'utf8', (err, data) => {
    let parsed;
    if (err)
    // 如果有错误产生则退出当前调用
      return callback(err);
    try {
      // 解析文件中的数据
      parsed = JSON.parse(data);
    } catch (err) {
      // 捕获解析中的错误，如果有错误产生，则进行错误处理
      return callback(err);
    }
    // 没有错误，调用回调
    callback(null, parsed);
  });
};
```

从上面的例子中我们注意到的细节是当我们想要正确地进行异常处理时，我们如何向 `callback` 传递参数。此外，当有错误产生时，我们使用了 `return` 语句，立即退出当前函数调用，避免进行下面的相关执行。

不可捕获的异常

从上述 `readJSON()` 函数，为了避免将任何异常抛到 `fs.readFile()` 的回调函数中捕获，我们对 `JSON.parse()` 周围放置一个 `try catch` 代码块。在异步回调中一旦出错，将抛出异常，并跳转到事件循环，不把错误传播到下一个回调函数去。

在 `Node.js` 中，这是一个不可恢复的状态，应用程序会关闭，并将错误打印到标准输出中。为了证明这一点，我们尝试从之前定义的 `readJSON()` 函数中删除 `try catch` 代码块：

```
const fs = require('fs');

function readJSONThrows(filename, callback) {
  fs.readFile(filename, 'utf8', (err, data) => {
    if (err) {
      return callback(err);
    }
    // 假设parse的执行没有错误
    callback(null, JSON.parse(data));
  });
};
```

在上面的代码中，我们没有办法捕获到 `JSON.parse` 产生的异常，如果我们尝试传递一个非标准 `JSON` 格式的文件，将会抛出以下错误：

```
➜ 07_callback_propagating_errors git:(master) ✗ node readJSON.js
undefined:1
{hello:"world"}
 ^

SyntaxError: Unexpected token h in JSON at position 1
    at JSON.parse (<anonymous>)
    at fs.readFile (/Users/encounter/workspace/Node.js_Design_Patterns_Second_Edition_Code/Chapter02/
07_callback_propagating_errors/readJSON.js:10:25)
    at tryToString (fs.js:513:3)
    at FSReqWrap.readFileAfterClose [as oncomplete] (fs.js:501:12)
```

```
SyntaxError: Unexpected token d
at Object.parse (native)
at [...]
at fs.js:266:14
at Object.oncomplete (fs.js:107:15)
```

现在，如果我们看看前面的错误栈跟踪，我们将看到它从 `fs` 模块的某处开始，恰好从本地 `API` 完成文件读取返回到 `fs.readFile()` 函数，通过事件循环。这些信息都很清楚地显示给我们，异常从我们的回调传入堆栈，然后直接进入事件循环，最终被捕获并抛出到控制台中。 这也意味着使用 `try catch` 代码块包装对 `readJSONThrows()` 的调用将不起作用，因为块所在的堆栈与调用回调的堆栈不同。以下代码显示了我们刚才描述的相反的情况：

```
try {
  readJSONThrows('nonJSON.txt', function(err, result) {
    // ...
  });
} catch (err) {
  console.log('This will not catch the JSON parsing exception');
}
```

前面的 `catch` 语句将永远不会收到 `JSON` 解析异常，因为它将返回到抛出异常的堆栈。我们刚刚看到堆栈在事件循环中结束，而不是触发异步操作的功能。 如前所述，应用程序在异常到达事件循环的那一刻中止，然而，我们仍然有机会在应用程序终止之前执行一些清理或日志记录。事实上，当这种情况发生时， `Node.js` 会在退出进程之前发出一个名为 `uncaughtException` 的特殊事件。以下代码显示了一个示例用例：

```
process.on('uncaughtException', (err) => {
  console.error('This will catch at last the ' +
    'JSON parsing exception: ' + err.message);
  // Terminates the application with 1 (error) as exit code:
  // without the following line, the application would continue
  process.exit(1);
});
```

```
➜ 07_callback_propagating_errors git:(master) ✗ node readJSON.js
This will catch at last the JSON parsing exception:
```

重要的是，未被捕获的异常会使应用程序处于不能保证一致的状态，这可能导致不可预见的问题。例如，可能还有不完整的 `I/O` 请求运行或关闭可能会变得不一致。这就是为什么总是建议，特别是在生产环境中，在接收到未被捕获的异常之后写上述代码进行错误日志记录。

模块系统及相关模式

模块不仅是构建大型应用的基础，其主要机制是封装内部实现、方法与变量，通过接口。在本节中，我们将介绍 `Node.js` 的模块系统及其最常见的使用模式。

关于模块

`JavaScript` 的主要问题之一是没有命名空间。在全局范围内运行的程序会污染全局命名空间，造成相关变量、数据、方法名的冲突。解决这个问题的技术称为模块模式，看下列代码：

```
const module = (() => {
  const privateFoo = () => {
    // ...
  };
  const privateBar = [];
  const exported = {
    publicFoo: () => {
      // ...
    },
    publicBar: () => {
      // ...
    }
  };
  return exported;
})();
console.log(module);
```

此模式利用自执行匿名函数实现模块，仅导出旨希望被公开调用的部分。在上面的代码中，模块变量只包含导出的 `API` ，而其余的模块内容实际上从外部访问不到。我们将在稍后看到，这种模式背后的想法被用作 `Node.js` 模块系统的基础。

Node.js模块相关解释

`CommonJS` 是一个旨在规范 `JavaScript` 生态系统的组织，他们提出了 `CommonJS模块规范` 。 `Node.js` 在此规范之上构建了其模块系统，并添加了一些自定义的扩展。为了描述它的工作原理，我们可以通过这样一个例子解释模块模式，每个模块都在私有命名空间下运行，这样模块内定义的每个变量都不会污染全局命名空间。

自定义模块系统

为了解释模块系统的远离，让我们从头开始构建一个类似的模块系统。下面的代码创建一个模仿 `Node.js` 原始 `require()` 函数的功能。

我们先创建一个加载模块内容的函数，将其包装到一个私有的命名空间内：

```
function loadModule(filename, module, require) {
  const wrappedSrc = `(function(module, exports, require) {
         ${fs.readFileSync(filename, 'utf8')}
       })(module, module.exports, require);`;
  eval(wrappedSrc);
}
```

模块的源代码被包装到一个函数中，如同自执行匿名函数那样。这里的区别在于，我们将一些固有的变量传递给模块，特指 `module` ， `exports` 和 `require` 。注意导出模块的参数是 `module.exports` 和 `exports` ，后面我们将再讨论。

请记住，这只是一个例子，在真实项目中可不要这么做。诸如 `eval()` 或vm模块有可能导致一些安全性的问题，它人可能利用漏洞来进行注入攻击。我们应该非常小心地使用甚至完全避免使用 `eval` 。

我们现在来看模块的接口、变量等是如何被 `require()` 函数引入的：

```
const require = (moduleName) => {
  console.log(`Require invoked for module: ${moduleName}`);
  const id = require.resolve(moduleName);
  // 是否命中缓存
  if (require.cache[id]) {
    return require.cache[id].exports;
  }
  // 定义module
  const module = {
    exports: {},
    id: id
  };
  // 新模块引入，存入缓存
  require.cache[id] = module;
  // 加载模块
  loadModule(id, module, require);
  // 返回导出的变量
  return module.exports;
};
require.cache = {};
require.resolve = (moduleName) => {
  /* 通过模块名作为参数resolve一个完整的模块 */
};
```

上面的函数模拟了用于加载模块的原生 `Node.js` 的 `require()` 函数的行为。当然，这只是一个 `demo` ，它并不能准确且完整地反映 `require()` 函数的真实行为，但是为了更好地理解 `Node.js` 模块系统的内部实现，定义模块和加载模块。我们的自制模块系统的功能如下：

- 模块名称被作为参数传入，我们首先做的是找寻模块的完整路径，我们称之为 `id` 。 `require.resolve()` 专门负责这项功能，它通过一个特定的解析算法实现相关功能（稍后将讨论）。
- 如果模块已经被加载，它应该存在于缓存。在这种情况下，我们立即返回缓存中的模块。
- 如果模块尚未加载，我们将首次加载该模块。创建一个模块对象，其中包含一个使用空对象字面值初始化的 `exports` 属性。该属性将被模块的代码用于导出该模块的公共 `API` 。
- 缓存首次加载的模块对象。
- 模块源代码从其文件中读取，代码被导入，如前所述。我们通过 `require()` 函数向模块提供我们刚刚创建的模块对象。该模块通过操作或替换 `module.exports` 对象来导出其公共API。
- 最后，将代表模块的公共 `API` 的 `module.exports` 的内容返回给调用者。

正如我们所看到的，`Node.js` 模块系统的原理并不是想象中那么高深，只不过是通过我们一系列操作来创建和导入导出模块源代码。

定义一个模块

通过查看我们的自定义 `require()` 函数的工作原理，我们现在既然已经知道如何定义一个模块。再来看下面这个例子：

```
// 加载另一个模块
const dependency = require('./anotherModule');
// 模块内的私有函数
function log() {
  console.log(`Well done ${dependency.username}`);
}
// 通过导出API实现共有方法
module.exports.run = () => {
  log();
};
```

需要注意的是模块内的所有内容都是私有的，除非它被分配给 `module.exports` 变量。然后，当使用 `require()` 加载模块时，缓存并返回此变量的内容。

定义全局变量

即使在模块中声明的所有变量和函数都在其本地范围内定义，仍然可以定义全局变量。事实上，模块系统公开了一个名为 `global` 的特殊变量。分配给此变量的所有内容将会被定义到全局环境下。

注意：污染全局命名空间是不好的，并且没有充分运用模块系统的优势。所以，只有真的需要使用全局变量，才去使用它。

module.exports和exports

对于许多还不熟悉 `Node.js` 的开发人员而言，他们最容易混淆的是 `exports` 和 `module.exports` 来导出公共 `API` 的区别。变量 `export` 只是对 `module.exports` 的初始值的引用;我们已经看到，`exports` 本质上在模块加载之前只是一个简单的对象。

这意味着我们只能将新属性附加到导出变量引用的对象，如以下代码所示：

```
exports.hello = () => {
  console.log('Hello');
}
```

重新给 `exports` 赋值并不会有任何影响，因为它并不会因此而改变 `module.exports` 的内容，它只是改变了该变量本身。因此下列代码是错误的：

```
exports = () => {
  console.log('Hello');
}
```

如果我们想要导出除对象之外的内容，比如函数，我们可以给 `module.exports` 重新赋值：

```
module.exports = () => {
  console.log('Hello');
}
```

require函数是同步的

另一个重要的细节是上述我们写的 `require()` 函数是同步的，它使用了一个较为简单的方式返回了模块内容，并且不需要回调函数。因此，对于 `module.exports` 也是同步的，例如，下列的代码是不正确的：

```
setTimeout(() => {
  module.exports = function() {
    // ...
  };
}, 100);
```

通过这种方式导出模块会对我们定义模块产生重要的影响，因为它限制了我们同步定义并使用模块的方式。这实际上是为什么核心 `Node.js` 库提供同步 `API` 以代替异步 `API` 的最重要的原因之一。

如果我们需要定义一个需要异步操作来进行初始化的模块，我们也可以随时定义和导出需要我们异步初始化的模块。但是这样定义异步模块我们并不能保证 `require()` 后可以立即使用，在第九章，我们将详细分析这个问题，并提出一些模式来优化解决这个问题。

实际上，在早期的 `Node.js` 中，曾经有一个异步版本的 `require()` ，但由于它对初始化时间和异步 `I/O` 的性能有巨大影响，很快这个 `API` 就被删除了。

resolve算法

`依赖地狱` 描述了软件的依赖于不同版本的软件包的依赖关系， `Node.js` 通过加载不同版本的模块来解决这个问题，具体取决于模块的加载位置。而都是由 `npm` 来完成的，相关算法被称作 `resolve算法` ，被用到 `require()` 函数中。

现在让我们快速概述一下这个算法。如下所述，`resolve()` 函数将一个模块名称（`moduleName`）作为输入，并返回模块的完整路径。然后，该路径用于加载其代码，并且还可以唯一地标识模块。`resolve算法` 可以分为以下三种规则：

- 文件模块：如果 `moduleName` 以 `/` 开头，那么它已经被认为是模块的绝对路径。如果以 `./` 开头，那么 `moduleName` 被认为是相对路径，它是从使用 `require` 的模块的位置开始计算的。
- 核心模块：如果 `moduleName` 不以 `/` 或 `./` 开头，则算法将首先尝试在核心Node.js模块中进行搜索。
- 模块包：如果没有找到匹配 `moduleName` 的核心模块，则搜索在当前目录下的 `node_modules`，如果没有搜索到 `node_modules`，则会往上层目录继续搜索 `node_modules`，直到它到达文件系统的根目录。

对于文件和包模块，单个文件和目录也可以匹配到 `moduleName`。特别地，算法将尝试匹配以下内容：

- `<moduleName>.js`
- `<moduleName>/index.js`
- 在 `<moduleName>/package.json` 的 `main` 值下声明的文件或目录

resolve算法的具体文档

`node_modules` 目录实际上是 `npm` 安装每个包并存放相关依赖关系的地方。这意味着，基于我们刚刚描述的算法，每个包都有自身的私有依赖关系。例如，看以下目录结构：

```
myApp
├── foo.js
└── node_modules
    ├── depA
    │   └── index.js
    └── depB
        │
        ├── bar.js
        ├── node_modules
        ├── depA
        │    └── index.js
        └── depC
             ├── foobar.js
             └── node_modules
                 └── depA
                     └── index.js
```

在前面的例子中，`myApp`，`depB` 和 `depC` 都依赖于 `depA`;然而，他们都有自己的私有依赖的版本！按照解析算法的规则，使用 `require('depA')` 将根据需要的模块加载不同的文件，如下：

- 在 `/myApp/foo.js` 中调用的 `require('depA')` 会加

  载 `/myApp/node_modules/depA/index.js`
- 在 `/myApp/node_modules/depB/bar.js` 中调用的 `require('depA')` 会加载 `/myApp/node_modules/depB/node_modules/depA/index.js`
- 在 `/myApp/node_modules/depC/foobar.js` 中调用的 `require('depA')` 会加载 `/myApp/node_modules/depC/node_modules/depA/index.js`

`resolve算法` 是 `Node.js` 依赖关系管理的核心部分，它的存在使得即便应用程序拥有成百上千包的情况下也不会出现冲突和版本不兼容的问题。

当我们调用 `require()` 时，解析算法对我们是透明的。然而，仍然可以通过调用 `require.resolve()` 直接由任何模块使用。

模块缓存

每个模块只会在它第一次引入的时候加载，此后的任意一次 `require()` 调用均从之前缓存的版本中取得。通过查看我们之前写的自定义的 `require()` 函数，可以看到缓存对于性能提升至关重要，此外也具有一些其它的优势，如下：

- 使得模块依赖关系的重复利用成为可能
- 从某种程度上保证了在从给定的包中要求相同的模块时总是返回相同的实例，避免了冲突

模块缓存通过 `require.cache` 变量查看，因此如果需要，可以直接访问它。在实际运用中的例子是通过删除 `require.cache` 变量中的相对键来使某个缓存的模块无效，这是在测试过程中非常有用，但在正常情况下会十分危险。

循环依赖

许多人认为循环依赖是 `Node.js` 内在的设计问题，但在真实项目中真的可能发生，所以我们至少知道如何在 `Node.js` 中使得循环依赖有效。再来看我们自定义的 `require()` 函数，我们可以立即看到其工作原理和注意事项。

看下面这两个模块：

- 模块 `a.js`

```
exports.loaded = false;
const b = require('./b');
module.exports = {
  bWasLoaded: b.loaded,
  loaded: true
};
```

- 模块 `b.js`

```
exports.loaded = false;
const a = require('./a');
module.exports = {
  aWasLoaded: a.loaded,
  loaded: true
};
```

然后我们在 `main.js` 中写以下代码：

```
const a = require('./a');
const b = require('./b');
console.log(a);
console.log(b);
```

执行上述代码，会打印以下结果：

```
{
  bWasLoaded: true,
  loaded: true
}
{
  aWasLoaded: false,
  loaded: true
}
```

```
➜ 11_module_circular_dependency git:(master) ✗ node main.js
{ bWasLoaded: true, loaded: true }
{ aWasLoaded: false, loaded: true }
```

这个结果展现了循环依赖的处理顺序。虽然 `a.js` 和 `b.js` 这两个模块都在主模块需要的时候完全初始化，但是当从 `b.js` 加载时， `a.js` 模块是不完整的。特别，这种状态会持续到 `b.js` 加载完毕的那一刻。这种情况我们应该引起注意，特别要确认我们在 `main.js` 中两个模块所需的顺序。

这是由于模块 `a.js` 将收到一个不完整的版本的 `b.js` 。我们现在明白，如果我们失去了首先加载哪个模块的控制，如果项目足够大，这可能会很容易发生循环依赖。

关于循环引用的文档

简单说就是，为了防止模块载入的死循环， `Node.js` 在模块第一次载入后会把它的结果进行缓存，下一次再对它进行载入的时候会直接从缓存中取出结果。所以在这种循环依赖情形下，不会有死循环，但是却会因为缓存造成模块没有按照我们预想的那样被导出（ `export` ，详细的案例分析见下文）。

官网给出了三个模块还不是循环依赖最简单的情形。实际上，两个模块就可以很清楚的表达出这种情况。根据递归的思想，解决了最简单的情形，这一类任意大小规模的问题也就解决了一半（另一半还需要探明随着问题规模增长，问题的解将会如何变化）。

`JavaScript` 作为一门解释型的语言，上面的打印输出清晰的展示出了程序运行的轨迹。在这个例子中， `a.js` 首先 `require` 了 `b.js` , 程序进入 `b.js` ，在 `b.js` 中第一行又 `require` 了 `a.js` 。

如前文所述，为了避免无限循环的模块依赖，在 `Node.js` 运行 `a.js` 之后，它就被缓存了，但需要注意的是，此时缓存的仅仅是一个未完工的 `a.js` （**an unfinished copy of the a.js**）。所以在 `b.js` 中 `require` 了 `a.js` 时，得到的仅仅是缓存中一个未完工的 `a.js` ，具体来说，它并没有明确被导出的具体内容（ `a.js` 尾端）。所以 `b.js` 中输出的 `a` 是一个空对象。

之后， `b.js` 顺利执行完，回到 `a.js` 的 `require` 语句之后，继续执行完成。

模块定义模式

模块系统除了自带处理依赖关系的机制之外，最常见的功能就是定义 `API` 。对于定义 `API` ，主要需要考虑私有和公共功能之间的平衡。其目的是最大化信息隐藏内部实现和暴露的 `API` 可用性，同时将这些与可扩展性和代码重用性进行平衡。

在本节中，我们将分析一些在 `Node.js` 中定义模块的最流行模式;每个模块都保证了私有变量的透明，可扩展性和代码重用。

命名导出

暴露公共 `API` 的最基本方法是使用命名导出，其中包括将我们想要公开的所有值分配给由 `export` （或 `module.exports` ）引用的对象的属性。以这种方式，生成的导出对象将成为一组相关功能的容器或命名空间。

看下面代码，是此模式的实现：

```
//file logger.js
exports.info = (message) => {
  console.log('info: ' + message);
};
exports.verbose = (message) => {
  console.log('verbose: ' + message);
};
```

导出的函数随后作为引入其的模块的属性使用，如下面的代码所示：

```
// file main.js
const logger = require('./logger');
logger.info('This is an informational message');
logger.verbose('This is a verbose message');
```

大多数 `Node.js` 模块使用这种定义。

`CommonJS` 规范仅允许使用 `exports` 变量来公开 `public` 成员。因此，命名的导出模式是唯一与 `CommonJS` 规范兼容的模式。使用 `module.exports` 是 `Node.js` 提供的一个扩展，以支持更广泛的模块定义模式。

函数导出

最流行的模块定义模式之一包括将整个 `module.exports` 变量重新分配给一个函数。它的主要优点是它只暴露了一个函数，为模块提供了一个明确的入口点，使其更易于理解和使用，它也很好地展现了单一职责原则。这种定义模块的方法在社区中也被称为 `substack模式` ，在以下示例中查看此模式：

```
// file logger.js
module.exports = (message) => {
  console.log(`info: ${message}`);
};
```

该模式也可以将导出的函数用作其他公共 `API` 的命名空间。这是一个非常强大的组合，因为它仍然给模块一个单独的入口点（ `exports` 的主函数）。这种方法还允许我们公开具有次要或更高级用例的其他函数。以下代码显示了如何使用导出的函数作为命名空间来扩展我们之前定义的模块：

```
module.exports.verbose = (message) => {
  console.log(`verbose: ${message}`);
};
```

这段代码演示了如何调用我们刚才定义的模块：

```
// file main.js
const logger = require('./logger');
logger('This is an informational message');
logger.verbose('This is a verbose message');
```

虽然只是导出一个函数也可能是一个限制，但实际上它是一个完美的方式，把重点放在一个单一的函数，它代表着这个模块最重要的一个功能，同时使得内部私有变量属性更加透明，而只是暴露导出函数本身的属性。

`Node.js` 的模块化鼓励我们遵循采用单一职责原则（ `SRP` ）：每个模块应该对单个功能负责，该职责应完全由该模块封装，以保证复用性。

注意，这里讲的 `substack模式` ，就是通过仅导出一个函数来暴露模块的主要功能。使用导出的函数作为命名空间来导出别的次要功能。

构造器(类)导出

导出构造函数的模块是导出函数的模块的特例。其不同之处在于，使用这种新模式，我们允许用户使用构造函数创建新的实例，但是我们也可以扩展其原型并创建新类（继承）。以下是此模式的示例：

```
// file logger.js
function Logger(name) {
  this.name = name;
}
Logger.prototype.log = function(message) {
  console.log(`[${this.name}] ${message}`);
};
Logger.prototype.info = function(message) {
  this.log(`info: ${message}`);
};
Logger.prototype.verbose = function(message) {
  this.log(`verbose: ${message}`);
};
module.exports = Logger;
```

我们通过以下方式使用上述模块：

```
// file main.js
const Logger = require('./logger');
const dbLogger = new Logger('DB');
dbLogger.info('This is an informational message');
const accessLogger = new Logger('ACCESS');
accessLogger.verbose('This is a verbose message');
```

通过 `ES2015` 的 `class` 关键字语法也可以实现相同的模式：

```
class Logger {
  constructor(name) {
    this.name = name;
  }
  log(message) {
    console.log(`[${this.name}] ${message}`);
  }
  info(message) {
    this.log(`info: ${message}`);
  }
  verbose(message) {
    this.log(`verbose: ${message}`);
  }
}
module.exports = Logger;
```

鉴于 `ES2015` 的类只是原型的语法糖，该模块的使用将与其基于原型和构造函数的方案完全相同。

导出构造函数或类仍然是模块的单个入口点，但与 `substack模式` 比起来，它暴露了更多的模块内部结构。然而，另一方面，当想要扩展该模块功能时，我们可以更加方便。

这种模式的变种包括对不使用 `new` 的调用。这个小技巧让我们将我们的模块用作工厂。看下列代码：

```
function Logger(name) {
  if (!(this instanceof Logger)) {
    return new Logger(name);
  }
  this.name = name;
};
```

其实这很简单：我们检查 `this` 是否存在，并且是 `Logger` 的一个实例。如果这些条件中的任何一个都为 `false` ，则意味着 `Logger()` 函数在不使用 `new` 的情况下被调用，然后继续正确创建新实例并将其返回给调用者。这种技术允许我们将模块也用作工厂：

```
// file logger.js
const Logger = require('./logger');
const dbLogger = Logger('DB');
accessLogger.verbose('This is a verbose message');
```

`ES2015` 的 `new.target` 语法从 `Node.js 6` 开始提供了一个更简洁的实现上述功能的方法。该利用公开了 `new.target` 属性，该属性是所有函数中可用的 `元属性` ，如果使用 `new` 关键字调用函数，则在运行时计算结果为 `true` 。我们可以使用这种语法重写工厂：

```
function Logger(name) {
  if (!new.target) {
    return new LoggerConstructor(name);
  }
  this.name = name;
}
```

这个代码完全与前一段代码作用相同，所以我们可以说 `ES2015` 的 `new.target` 语法糖使得代码更加可读和自然。

实例导出

我们可以利用 `require()` 的缓存机制来轻松地定义具有从构造函数或工厂创建的状态的有状态实例，可以在不同模块之间共享。以下代码显示了此模式的示例：

```
//file logger.js
function Logger(name) {
  this.count = 0;
  this.name = name;
}
Logger.prototype.log = function(message) {
  this.count++;
  console.log('[' + this.name + '] ' + message);
};
module.exports = new Logger('DEFAULT');
```

这个新定义的模块可以这么使用：

```
// file main.js
const logger = require('./logger');
logger.log('This is an informational message');
```

因为模块被缓存，所以每个需要 `Logger` 模块的模块实际上总是会检索该对象的相同实例，从而共享它的状态。这种模式非常像创建单例。然而，它并不保证整个应用程序的实例的唯一性，因为它发生在传统的单例模式中。在分析解析算法时，实际上已经看到，一个模块可能会多次安装在应用程序的依赖关系树中。这导致了同一逻辑模块的多个实例，所有这些实例都运行在同一个 `Node.js` 应用程序的上下文中。在第7章中，我们将分析导出有状态的实例和一些可替代的模式。

我们刚刚描述的模式的扩展包括 `exports` 用于创建实例的构造函数以及实例本身。这允许用户创建相同对象的新实例，或者如果需要也可以扩展它们。为了实现这一点，我们只需要为实例分配一个新的属性，如下面的代码所示：

```
module.exports.Logger = Logger;
```

然后，我们可以使用导出的构造函数创建类的其他实例：

```
const customLogger = new logger.Logger('CUSTOM');
customLogger.log('This is an informational message');
```

从代码可用性的角度来看，这类似于将导出的函数用作命名空间，该模块导出一个对象的默认实例，这是我们大部分时间使用的功能，而更多的高级功能（如创建新实例或扩展对象的功能）仍然可以通过较少的暴露属性来使用。

修改其他模块或全局作用域

一个模块甚至可以导出任何东西这可以看起来有点不合适;但是，我们不应该忘记一个模块可以修改全局范围和其中的任何对象，包括缓存中的其他模块。请注意，这些通常被认为是不好的做法，但是由于这种模式在某些情况下（例如测试）可能是有用和安全的，有时确实可以利用这一特性，这是值得了解和理解的。我们说一个模块可以修改全局范围内的其他模块或对象。它通常是指在运行时修改现有对象以更改或扩展其行为或应用的临时更改。

以下示例显示了我们如何向另一个模块添加新函数：

```
// file patcher.js
// ./logger is another module
require('./logger').customMessage = () => console.log('This is a
 new functionality');
```

编写以下代码：

```
// file main.js
require('./patcher');
const logger = require('./logger');
logger.customMessage();
```

在上述代码中，必须首先引入 `patcher` 程序才能使用 `logger` 模块。

上面的写法是很危险的。主要考虑的是拥有修改全局命名空间或其他模块的模块是具有副作用的操作。换句话说，它会影响其范围之外的实体的状态，这可能导致不可预测的后果，特别是当多个模块与相同的实体进行交互时。想象一下，有两个不同的模块尝试设置相同的全局变量，或者修改同一个模块的相同属性，效果可能是不可预测的（哪个模块胜出？），但最重要的是它会对在整个应用程序产生影响。

观察者模式

`Node.js` 中的另一个重要和基本的模式是观察者模式。与 `reactor模式` ，回调模式和模块一样，观察者模式是 `Node.js` 基础之一，也是使用许多 `Node.js` 核心模块和用户定义模块的基础。

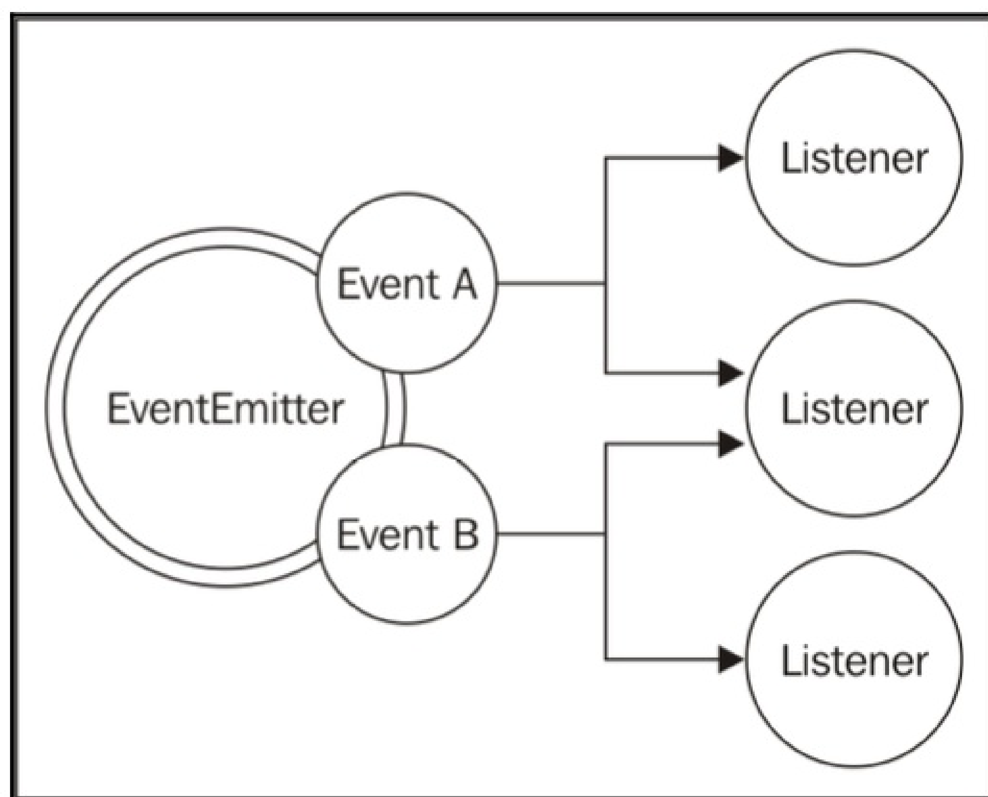
观察者模式是对 `Node.js` 的数据响应的理想解决方案，也是对回调的完美补充。我们给出以下定义：

发布者定义一个对象，它可以在其状态发生变化时通知一组观察者（或监听者）。

与回调模式的主要区别在于，主体实际上可以通知多个观察者，而传统的 `CPS` 风格的回调通常主体的结果只会传播给一个监听器。

EventEmitter类

在传统的面向对象编程中，观察者模式需要接口，具体类和层次结构。在 `Node.js` 中，都变得简单得多。观察者模式已经内置在核心模块中，可以通过 `EventEmitter` 类来实现。 `EventEmitter` 类允许我们注册一个或多个函数作为监听器，当特定的事件类型被触发时，它的回调将被调用，以通知其监听器。以下图像直观地解释了这个概念：



`EventEmitter` 是一个类（原型），它是从事件核心模块导出的。以下代码显示了如何获得对它的引用：

```
const EventEmitter = require('events').EventEmitter;
const eeInstance = new EventEmitter();
```

`EventEmitter` 的基本方法如下：

- `on(event，listener)` ：此方法允许您为给定的事件类型（ `String类型` ）注册一个新的侦听器（一个函数）
- `once(event, listener)` ：此方法注册一个新的监听器，然后在事件首次发布之后被删除
- `emit(event, [arg1], [...])` ：此方法会生成一个新事件，并提供其他参数以传递给侦听器
- `removeListener(event, listener)` ：此方法将删除指定事件类型的侦听器

所有上述方法将返回 `EventEmitter` 实例以允许链接。监听器函数 `function([arg1], [...])` ，所以它只是接受事件发出时提供的参数。在侦听器中，这是指 `EventEmitter` 生成事件的实例。 我们可以看到，一个监听器和一个传统的 `Node.js` 回调有很大的区别;特别地，第一个参数不是 `error` ，它是在调用时传递给 `emit()` 的任何数据。

创建和使用**EventEmitter**

我们来看看我们如何在实践中使用 `EventEmitter` 。最简单的方法是创建一个新的实例并立即使用它。以下代码显示了在文件列表中找到匹配特定正则的文件内容时，使用 `EventEmitter` 实现实时通知订阅者的功能：

```
const EventEmitter = require('events').EventEmitter;
const fs = require('fs');

function findPattern(files, regex) {
  const emitter = new EventEmitter();
  files.forEach(function(file) {
    fs.readFile(file, 'utf8', (err, content) => {
      if (err)
        return emitter.emit('error', err);
      emitter.emit('fileread', file);
      let match;
      if (match = content.match(regex))
        match.forEach(elem => emitter.emit('found', file, elem))
;
    });
  });
  return emitter;
}
```

由前面的函数 `EventEmitter` 处理将产生的三个事件：

- `fileread` 事件：当文件被读取时触发
- `found` 事件：当文件内容被正则匹配成功时触发
- `error` 事件：当读取文件出现错误时触发

下面看 `findPattern()` 函数是如何被触发的：

```
findPattern(['fileA.txt', 'fileB.json'], /hello \w+/g)
  .on('fileread', file => console.log(file + ' was read'))
  .on('found', (file, match) => console.log('Matched "' + match
+ '" in file ' + file))
  .on('error', err => console.log('Error emitted: ' + err.messag
e));
```

在前面的例子中，我们为 `EventPartern()` 函数创建的 `EventEmitter` 生成的每个事件类型注册了一个监听器。

错误传播

如果事件是异步发送的， `EventEmitter` 不能在异常情况发生时抛出异常，异常会在事件循环中丢失。相反，而是 `emit` 是发出一个称为错误的特殊事件， `Error对象` 通过参数传递。这正是我们在之前定义的 `findPattern()` 函数中正在做的。

对于错误事件，始终是最佳做法注册侦听器，因为 `Node.js` 会以特殊的方式处理它，并且如果没有找到相关联的侦听器，将自动抛出异常并退出程序。

让任意对象可观察

有时，直接通过 `EventEmitter` 类创建一个新的可观察的对象是不够的，因为原生 `EventEmitter` 类并没有提供我们实际运用场景的拓展功能。我们可以通过扩展 `EventEmitter` 类使一个通用对象可观察。

为了演示这个模式，我们试着在对象中实现 `findPattern()` 函数的功能，如下代码所示：

```
const EventEmitter = require('events').EventEmitter;
const fs = require('fs');
class FindPattern extends EventEmitter {
  constructor(regex) {
    super();
    this.regex = regex;
    this.files = [];
  }
  addFile(file) {
    this.files.push(file);
    return this;
  }
  find() {
    this.files.forEach(file => {
      fs.readFile(file, 'utf8', (err, content) => {
        if (err) {
          return this.emit('error', err);
        }
        this.emit('fileread', file);
        let match = null;
        if (match = content.match(this.regex)) {
          match.forEach(elem => this.emit('found', file, elem));
        }
      });
    });
    return this;
  }
}
```

我们定义的 `FindPattern` 类中运用了核心模块 `util` 提供的 `inherits()` 函数来扩展 `EventEmitter` 。以这种方式，它成为一个符合我们实际运用场景的可观察类。以下是其用法的示例：

```
const findPatternObject = new FindPattern(/hello \w+/);
findPatternObject
  .addFile('fileA.txt')
  .addFile('fileB.json')
  .find()
  .on('found', (file, match) => console.log(`Matched "${match}"
       in file ${file}`))
  .on('error', err => console.log(`Error emitted ${err.message}`
));
```

现在，通过继承 `EventEmitter` 的功能，我们现在可以看到 `FindPattern` 对象除了可观察外，还有一整套方法。 这在 `Node.js` 生态系统中是一个很常见的模式，例如，核心 `HTTP` 模块的 `Server` 对象定义

了 `listen()` ， `close()` ， `setTimeout()` 等方法，并且在内部它也继承自 `EventEmitter` 函数，从而允许它在收到新的请求、建立新的连接或者服务器关闭响应请求相关的事件。

扩展 `EventEmitter` 的对象的其他示例是 `Node.js` 流。我们将在第五章中更详细地分析 `Node.js` 的流。

同步和异步事件

与回调模式类似，事件也支持同步或异步发送。至关重要的是，我们决不应当在同一个 `EventEmitter` 中混合使用两种方法，但是在发布相同的事件类型时考虑同步或者异步显得至关重要，以避免产生因同步与异步顺序不一致导致的 `zalgo` 。

发布同步和异步事件的主要区别在于观察者注册的方式。当事件异步发布时，即使在 `EventEmitter` 初始化之后，程序也会注册新的观察者，因为必须保证此事件在事件循环下一周期之前不被触发。正如上边的 `findPattern()` 函数中的情况。它代表了大多数 `Node.js` 异步模块中使用的常用方法。

相反，同步发布事件要求在 `EventEmitter` 函数开始发出任何事件之前就得注册好观察者。看下面的例子：

```
const EventEmitter = require('events').EventEmitter;
class SyncEmit extends EventEmitter {
  constructor() {
    super();
    this.emit('ready');
  }
}
const syncEmit = new SyncEmit();
syncEmit.on('ready', () => console.log('Object is ready to be  u
sed'));
```

如果 `ready` 事件是异步发布的，那么上述代码将会正常运行，然而，由于事件是同步发布的，并且监听器在发送事件之后才被注册，所以结果不调用监听器，该代码将无法打印到控制台。

由于不同的应用场景，有时以同步方式使用 `EventEmitter` 函数是有意义的。因此，要清楚地突出我们的 `EventEmitter` 的同步和异步性，以避免产生不必要的错误和异常。

事件机制与回调机制的比较

在定义异步 `API` 时，常见的难点是检查是否使用 `EventEmitter` 的事件机制或仅接受回调函数。一般区分规则是这样的：当一个结果必须以异步方式返回时，应该使用回调函数，当需要结果不确定其方式时，应该使用事件机制来响应。

但是，由于这两者实在太相近，并且可能两种方式都能实现相同的应用场景，所以产生了许多混乱。以下列代码为例：

```
function helloEvents() {
  const eventEmitter = new EventEmitter();
  setTimeout(() => eventEmitter.emit('hello', 'hello world'), 100
);
  return eventEmitter;
}

function helloCallback(callback) {
  setTimeout(() => callback('hello world'), 100);
}
```

`helloEvents()` 和 `helloCallback()` 在其功能上可以被认为是等价的，第一个使用事件机制实现，第二个则使用回调来通知调用者，而将事件作为参数传递。但是真正区分它们的是可执行性，语义和要实现或使用的代码量。虽然我们不能给出一套确定性的规则来选择一种风格，但我们当然可以提供一些提示来帮助你做出决定。

相比于第一个例子，即观察者模式而言，回调函数在支持不同类型的事件时有一些限制。但是事实上，我们仍然可以通过将事件类型作为回调的参数传递，或者通过接受多个回调来区分多个事件。然而，这样做的话不能被认为是一个优雅的 `API` 。在这种情况下， `EventEmitter` 可以提供更好的接口和更精简的代码。

`EventEmitter` 更优秀的另一种应用场景是多次触发同一事件或不触发事件的情况。事实上，无论操作是否成功，一个回调预计都只会被调用一次。但有一种特殊情况是，我们可能不知道事件在哪个时间点触发，在这种情况下， `EventEmitter` 是首选。

最后，使用回调的 `API` 仅通知特定的回调，但是使用 `EventEmitter` 函数可以让多个监听器都接收到通知。

回调机制和事件机制结合使用

还有一些情况可以将事件机制和回调结合使用。特别是当我们导出异步函数时，这种模式非常有用。node-glob模块是该模块的一个示例。

```
glob(pattern, [options], callback)
```

该函数将一个文件名匹配模式作为第一个参数，后面两个参数分别为一组选项和一个回调函数，对于匹配到指定文件名匹配模式的文件列表，相关回调函数会被调用。同时，该函数返回 `EventEmitter` ，它展现了当前进程的状态。例如，当成功匹配文件名时可以实时发布 `match` 事件，当文件列表全部匹配完毕时可以实时发布 `end` 事件，或者该进程被手动中止时发布 `abort` 事件。看以下代码：

```
const glob = require('glob');
glob('data/*.txt', (error, files) => console.log(`All files foun
d: ${JSON.stringify(files)}`))
  .on('match', match => console.log(`Match found: ${match}`));
```

总结

在本章中，我们首先了解了同步和异步的区别。然后，我们探讨了如何使用回调机制和回调机制来处理一些基本的异步方案。我们还了解到两种模式之间的主要区别，何时比另一种模式更适合解决具体问题。我们只是迈向更先进的异步模式的第一步。

在下一章中，我们将介绍更复杂的场景，了解如何利用回调机制和事件机制来处理高级异步控制问题。

Asynchronous Control Flow Patterns with Callbacks

`Node.js` 这类语言习惯于同步的编程风格，其 `CPS` 风格和异步特性的 `API` 是其标准，对于新手来说可能难以理解。编写异步代码可能是一种不同的体验，尤其是对异步控制流而言。异步代码可能让我们难以预测在 `Node.js` 中执行语句的顺序。例如读取一组文件，执行一串任务，或者等待一组操作完成，都需要开发人员采用新的方法和技术，以避免最终编写出效率低下和不可维护的代码。一个常见的错误是回调地狱，代码量急剧上升又不可读，使得简单的程序也难以阅读和维护。在本章中，我们将看到如何通过使用一些规则和一些模式来避免回调，并编写干净、可管理的异步代码。我们将看到控制流库，如 `async` ，可以极大地简化我们的问题，提升我们的代码可读性，更易于维护。

异步编程的困难

`JavaScript` 中异步代码的顺序错乱无疑是很容易的。闭包和对匿名函数的定义可以使开发人员有更好的编程体验，而并不需要开发人员手动对异步操作进行管理和跳转。这是符合 `KISS` 原则的。简单且能保持异步代码控制流，让它在更短的时间内工作。但不幸的是，回调嵌套是以牺牲诸如模块性、可重用性和可维护性，增大整个函数的大小，导致糟糕的代码结构为代价的。大多数情况下，创建闭包在功能上是不需要的，但这更多是一种约束，而不是与异步编程相关的问题。认识到回调嵌套会使得我们的代码变得笨拙，然后根据最适合的解决方案采取相应的方法解决回调地狱，这是新手与专家的区别。

创建一个简单的**Web**爬虫

为了解释上述问题，我们创建了一个简单的Web爬虫，一个命令行应用，其接受一个 `URL` 为输入，然后可以把其内容下载到一个文件中。在下列代码中，我们会依赖以下两个 `npm` 库。

此外，我们还将引用一个叫做 `./utilities` 的本地模块。

我们的应用程序的核心功能包含在一个名为 `spider.js` 的模块中。如下所示，首先加载我们所需要的依赖包：

```
const request = require('request');
const fs = require('fs');
const mkdirp = require('mkdirp');
const path = require('path');
const utilities = require('./utilities');
```

接下来，我们将创建一个名为 `spider()` 的新函数，该函数接受 `URL` 为参数，并在下载过程完成时调用一个回调函数。

```
function spider(url, callback) {
  const filename = utilities.urlToFilename(url);
  fs.exists(filename, exists => {
    if (!exists) {
      console.log(`Downloading ${url}`);
      request(url, (err, response, body) => {
        if (err) {
          callback(err);
        } else {
          mkdirp(path.dirname(filename), err => {
            if (err) {
              callback(err);
            } else {
              fs.writeFile(filename, body, err => {
                if (err) {
                  callback(err);
                } else {
                  callback(null, filename, true);
                }
              });
            }
          });
        }
      });
    } else {
      callback(null, filename, false);
    }
  });
}
```

上述函数执行以下任务：

- 检查该 `URL` 的文件是否已经下载过，即验证相应文件是否已经被创建：

`fs.exists(filename, exists => ...`

- 如果文件还没有被下载，则执行下列代码进行下载操作：

`request(url, (err, response, body) => ...`

- 然后，我们需要确定目录下是否已经包含了该文件：

`mkdirp(path.dirname(filename), err => ...`

- 最后，我们把 `HTTP` 请求返回的报文主体写入文件系统：

`mkdirp(path.dirname(filename), err => ...`

要完成我们的 `Web爬虫` 应用程序，只需提供一个 `URL` 作为输入(在我们的例子中，我们从命令行参数中读取它)，我们只需调用 `spider()` 函数即可。

```
spider(process.argv[2], (err, filename, downloaded) => {
  if (err) {
    console.log(err);
  } else if (downloaded) {
    console.log(`Completed the download of "${filename}"`);
  } else {
    console.log(`"${filename}" was already downloaded`);
  }
});
```

现在，我们开始尝试运行 `Web爬虫` 应用程序，但是首先，确保已有 `utilities.js` 模块和 `package.json` 中的所有依赖包已经安装到你的项目中：

```
npm install
```

之后，我们执行我们这个爬虫模块来下载一个网页，使用以下命令：

```
node spider http://www.example.com
```

我们的 `Web爬虫` 应用程序要求在我们提供的 `URL` 中总是包含协议类型(例如， `http://`)。另外，不要期望 `HTML` 链接被重新编写，也不要期望下载像图片这样的资源，因为这只是一个简单的例子来演示异步编程是如何工作的。

```
➜ 01_web_spider git:(master) ✗ ls
README.txt    index.js     package.json utilities.js
➜ 01_web_spider git:(master) ✗ cnpm install
✔ Installed 3 packages
✔ Linked 53 latest versions
✔ Run 0 scripts
✔ All packages installed (57 packages installed from npm registry, used 2s, speed 550.84kB/s, json 56(111.88kB), tarball 1.22MB)
➜ 01_web_spider git:(master) ✗ node index.js http://www.baidu.com
Downloading http://www.baidu.com
Completed the download of "www.baidu.com.html"
```

回调地狱

看看我们的 `spider()` 函数，我们可以发现，尽管我们实现的算法非常简单，但是生成的代码有几个级别的缩进，而且很难读懂。使用阻塞式的同步 `API` 实现类似的功能是很简单的，而且很少有机会让它看起来如此错误。然而，使用异步 `CPS` 是另一回事，使用闭包可能会导致出现难以阅读的代码。

大量闭包和回调将代码转换成不可读的、难以管理的情况称为回调地狱。它是 `Node.js` 中最受认可和最严重的反模式之一。一般来说，对于 `JavaScript` 而言。受此问题影响的代码的典型结构如下:

```
asyncFoo(err => {
  asyncBar(err => {
    asyncFooBar(err => {
      //...
    });
  });
});
```

我们可以看到，用这种方式编写的代码是如何形成金字塔形状的，由于深嵌的原因导致的难以阅读，称为“末日金字塔”。

像前面的代码片段这样的代码最明显的问题是可读性差。由于嵌套太深，几乎不可能跟踪回调函数的结束位置和另一个回调函数开始的位置。

另一个问题是由每个作用域中使用的变量名的重叠引起的。通常，我们必须使用类似甚至相同的名称来描述变量的内容。最好的例子是每个回调接收到的错误参数。有些人经常尝试使用相同名称的变体来区分每个范围内的对象，例如，`error` 、 `err` 、 `err1` 、 `err2` 等等。另一些人则倾向于隐藏在范围中定义的变量，总是使用相同的名称。例如， `err` 。这两种选择都远非完美，而且会造成混淆，并增加导致 `bug` 的可能性。

此外，我们必须记住，虽然闭包在性能和内存消耗方面的代价很小。此外，它们还可以创建不易识别的内存泄漏，因为我们不应该忘记，由闭包引用的任何上下文变量都不会被垃圾收集所保留。

关于对于 `V8` 的闭包工作原理，可以参考Vyacheslav Egorov的博客文章。

如果我们看一下我们的 `spider()` 函数，我们会清楚地注意到它便是一个典型的回调地狱的场景，并且在这个函数中有我们刚才描述的所有问题。这正是我们将在本章中学习的模式和技巧所要解决的问题。

使用简单的JavaScript

既然我们已经遇到了第一个回调地狱的例子，我们知道我们应该避免什么。然而，在编写异步代码时，这并不是惟一的关注点。事实上，有几种情况下，控制一组异步任务的流需要使用特定的模式和技术，特别是如果我们只使用普通的 `JavaScript` 而没有任何外部库的帮助的情况下。例如，通过按顺序应用异步操作来遍历集合并不像在数组中调用 `forEach()` 那样简单，但实际上它需要一种类似于递归的技术。

在本节中，我们将学习如何避免回调地狱，以及如何使用简单的 `JavaScript` 实现一些最常见的控制流模式。

回调函数的准则

在编写异步代码时，要记住的第一个规则是在定义回调时不要滥用闭包。滥用闭包一时很爽，因为它不需要对诸如模块化和可重用性这样的问题进行额外的思考。但是，我们已经看到，这种做法弊大于利。大多数情况下，修复回调地狱问题并不需要任何库、花哨的技术或范式的改变，只是一些常识。

以下是一些基本原则，可以帮助我们更少的嵌套，并改进我们的代码的组织：

- 尽可能退出外层函数。根据上下文，使用 `return` 、 `continue` 或 `break` ，以便立即退出当前代码块，而不是使用 `if...else` 代码块。其他语句。这将有助于优化我们的代码结构。
- 为回调创建命名函数，避免使用闭包，并将中间结果作为参数传递。命名函数也会使它们在堆栈跟踪中更优雅。
- 代码尽可能模块化。并尽可能将代码分成更小的、可重用的函数。

回调调用的准则

为了展示上述原则，我们通过重构 `Web爬虫` 应用程序来说明。

对于第一步，我们可以通过删除 `else` 语句来重构我们的错误检查方式。这是在我们收到错误后立即从函数中返回。因此，看以下代码：

```
if (err) {
  callback(err);
} else {
  // 如果没有错误，执行该代码块
}
```

我们可以通过编写下面的代码来改进我们的代码结构：

```
if (err) {
  return callback(err);
}
// 如果没有错误，执行该代码块
```

有了这个简单的技巧，我们立即减少了函数的嵌套级别，它很简单，不需要任何复杂的重构。

在执行我们刚才描述的优化时，一个常见的错误是在调用回调函数之后忘记终止函数，即 `return` 。对于错误处理场景，以下代码是 `bug` 的典型来源：

```
if (err) {
  callback(err);
}
// 如果没有错误，执行该代码块
```

在这个例子中，即使在调用回调之后，函数的执行也会继续。那么避免这种情况的出现， `return` 语句是十分必要的。还要注意，函数返回的输出是什么并不重要，实际结果(或错误)是异步生成的，并传递给回调。异步函数的返回值通常被忽略。该属性允许我们编写如下的代码：

```
return callback(...);
```

否则我们必须拆成两条语句来写：

```
callback(...);
return;
```

接下来我们继续重构我们的 `spider()` 函数，我们可以尝试识别可复用的代码片段。例如，将给定字符串写入文件的功能可以很容易地分解为一个单独的函数：

```
function saveFile(filename, contents, callback) {
  mkdirp(path.dirname(filename), err => {
    if (err) {
      return callback(err);
    }
    fs.writeFile(filename, contents, callback);
  });
}
```

遵循同样的原则，我们可以创建一个名为 `download()` 的通用函数，它将 `URL` 和 `文件名` 作为输入，并将 `URL` 的内容下载到给定的文件中。在内部，我们可以使用前面创建的 `saveFile()` 函数。

```
function download(url, filename, callback) {
  console.log(`Downloading ${url}`);
  request(url, (err, response, body) => {
    if (err) {
      return callback(err);
    }
    saveFile(filename, body, err => {
      if (err) {
        return callback(err);
      }
      console.log(`Downloaded and saved: ${url}`);
      callback(null, body);
    });
  });
}
```

最后，修改我们的 `spider()` 函数：

```
function spider(url, callback) {
  const filename = utilities.urlToFilename(url);
  fs.exists(filename, exists => {
    if (exists) {
      return callback(null, filename, false);
    }
    download(url, filename, err => {
      if (err) {
        return callback(err);
      }
      callback(null, filename, true);
    })
  });
}
```

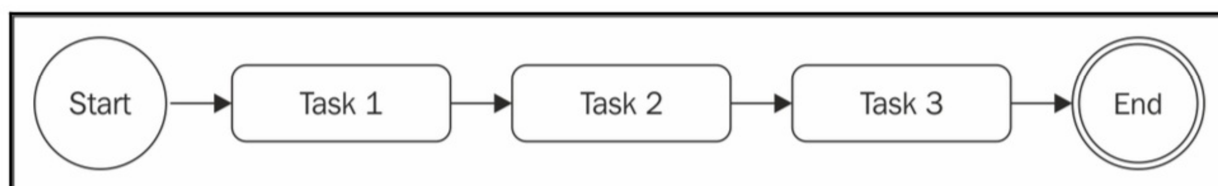
`spider()` 函数的功能和接口仍然是完全相同的，改变的仅仅是代码的组织方式。通过应用上述基本原则，我们能够极大地减少代码的嵌套，同时增加了它的可重用性和可测试性。实际上，我们可以考虑导出 `saveFile()` 和 `download()` ，这样我们就可以在其他模块中重用它们。这也使我们能够更容易地测试他们的功能。

我们在这一节中进行的重构清楚地表明，大多数时候，我们所需要的只是一些规则，并确保我们不滥用闭包和匿名函数。它的工作非常出色，只需最少的工作量，并且只使用原始的 `JavaScript` 。

顺序执行

现在开始探寻异步控制流的执行顺序，我们会通过开始分析一串异步代码来探寻其控制流。

按顺序执行一组任务意味着一次一个接一个地运行它们。执行顺序很重要，必须保证其正确性，因为列表中一个任务的结果可能会影响下一个任务的执行。下图说明了这个概念：



上述异步控制流有一些不同的变化：

- 按顺序执行一组已知任务，无需链接或传递执行结果
- 使用任务的输出作为下一个输入（也称为 `chain` ， `pipeline` ，或者 `waterfall` ）
- 在每个元素上运行异步任务时迭代一个集合，一个元素接一个元素

对于顺序执行而言，尽管在使用直接样式阻塞 `API` 实现很简单，但通常情况下使用 `异步CPS` 时会导致回调地狱问题。

按顺序执行一组已知的任务

在上一节中实现 `spider()` 函数时，我们已经遇到了顺序执行的问题。通过研究如下方式，我们可以更好地控制异步代码。以该代码为准则，我们可以用以下模式来解决上述问题：

```
function task1(callback) {
  asyncOperation(() => {
    task2(callback);
  });
}

function task2(callback) {
  asyncOperation(result() => {
    task3(callback);
  });
}

function task3(callback) {
  asyncOperation(() => {
    callback(); //finally executes the callback
  });
}

task1(() => {
  //executed when task1, task2 and task3 are completed
  console.log('tasks 1, 2 and 3 executed');
});
```

上述模式显示了在完成一个异步操作后，再调用下一个异步操作。该模式强调任务的模块化，并且避免在处理异步代码使用闭包。

顺序迭代

我们前面描述的模式如果我们预先知道要执行什么和有多少个任务，这些模式是完美的。这使我们能够对序列中下一个任务的调用进行硬编码，但是如果要对集合中的每个项目执行异步操作，会发生什么？在这种情况下，我们不能对任务序列进行硬编码。相反的是，我们必须动态构建它。

Web爬虫版本2

为了显示顺序迭代的例子，让我们为 `Web爬虫` 应用程序引入一个新功能。我们现在想要递归地下载网页中的所有链接。要做到这一点，我们将从页面中提取所有链接，然后按顺序逐个地触发我们的 `Web爬虫` 应用程序。

第一步是修改我们的 `spider()` 函数，以便通过调用一个名为 `spiderLinks()` 的函数触发页面所有链接的递归下载。

此外，我们现在尝试读取文件，而不是检查文件是否已经存在，并开始爬取其链接。这样，我们就可以恢复中断的下载。最后还有一个变化是，我们确保我们传递的参数是最新的，还要限制递归深度。结果代码如下：

```
function spider(url, nesting, callback) {
  const filename = utilities.urlToFilename(url);
  fs.readFile(filename, 'utf8', (err, body) => {
    if (err) {
      if (err.code! == 'ENOENT') {
        return callback(err);
      }
      return download(url, filename, (err, body) => {
        if (err) {
          return callback(err);
        }
        spiderLinks(url, body, nesting, callback);
      });
    }
    spiderLinks(url, body, nesting, callback);
  });
}
```

爬取链接

现在我们可以创建这个新版本的 `Web爬虫` 应用程序的核心，即 `spiderLinks()` 函数，它使用顺序异步迭代算法下载 `HTML` 页面的所有链接。注意我们在下面的代码块中定义的方式：

```
function spiderLinks(currentUrl, body, nesting, callback) {
  if(nesting === 0) {
    return process.nextTick(callback);
  }

  let links = utilities.getPageLinks(currentUrl, body); //[1]
  function iterate(index) { //[2]
    if(index === links.length) {
      return callback();
    }

    spider(links[index], nesting - 1, function(err) { //[3]
      if(err) {
        return callback(err);
      }
      iterate(index + 1);
    });
  }
  iterate(0); //[4]
}
```

从这个新功能中的重要步骤如下：

1. 我们使用 `utilities.getPageLinks()` 函数获取页面中包含的所有链接的列表。此函数仅返回指向相同主机名的链接。
2. 我们使用一个称为 `iterate()` 的本地函数来遍历链接，该函数需要下一个链接的索引进行分析。在这个函数中，我们首先要检查索引是否等于链接数组的长度，如果等于则是迭代完成，在这种情况下我们立即调用 `callback()` 函数，因为这意味着我们处理了所有的项目。
3. 这时，处理链接已准备就绪。我们通过递归调用 `spider()` 函数。
4. 作为 `spiderLinks()` 函数的最后一步也是最重要的一步，我们通过调用 `iterate(0)` 来开始迭代。

我们刚刚提出的算法允许我们通过顺序执行异步操作来迭代数组，在我们的例子中是 `spider()` 函数。

我们现在可以尝试这个新版本的 `Web爬虫` 应用程序，并观看它一个接一个地递归地下载网页的所有链接。要中断这个过程，如果有很多链接可能需要一段时间，请记住我们可以随时使用 `Ctrl + C` 。如果我们决定恢复它，我们可以通过启动 `Web爬虫` 应用程序并提供与上次结束时相同的 `URL` 来恢复执行。

现在我们的网络 `Web爬虫` 应用程序可能会触发整个网站的下载，请仔细考虑使用它。例如，不要设置高嵌套级别或离开爬虫运行超过几秒钟。用数千个请求重载服务器是不道德的。在某些情况下，这也被认为是非法的。需要考虑后果！

迭代模式

我们之前展示的 `spiderLinks()` 函数的代码是一个清楚的例子，说明了如何在应用异步操作时迭代集合。我们还可以注意到，这是一种可以适应任何其他情况的模式，我们需要在集合的元素或通常的任务列表上按顺序异步迭代。该模式可以推广如下：

```
function iterate(index) {
  if (index === tasks.length) {
    return finish();
  }
  const task = tasks[index];
  task(function() {
    iterate(index + 1);
  });
}

function finish() {
  // 迭代完成的操作
}

iterate(0);
```

注意到，如果 `task()` 是同步操作，这些类型的算法变得真正递归。在这种情况下，可能造成调用栈的溢出。

我们刚刚提出的模式是非常强大的，因为它可以适应几种情况。例如，我们可以映射数组的值，或者我们可以将迭代的结果传递给迭代中的下一个，以实现一个reduce算法，如果满足特定的条件，我们可以提前退出循环，或者甚至可以迭代无限数量的元素。

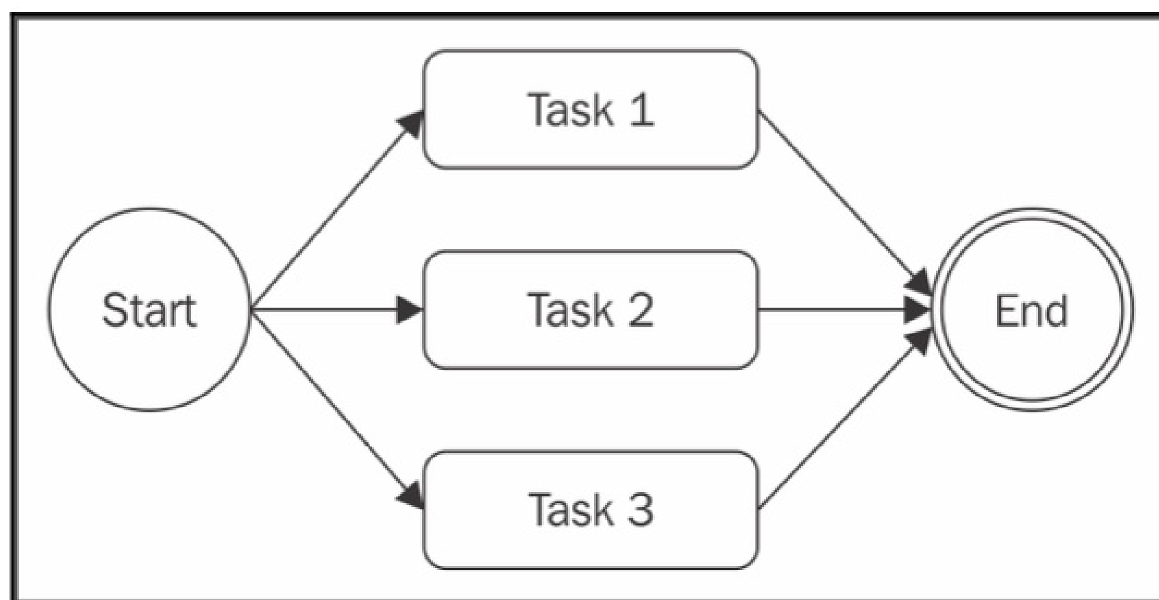
我们还可以选择将解决方案进一步推广：

```
iterateSeries(collection, iteratorCallback, finalCallback);
```

通过创建一个名为 `iterator` 的函数来执行任务列表，该函数调用集合中的下一个可执行的任务，并确保在当前任务完成时调用迭代器结束的回调函数。

并行

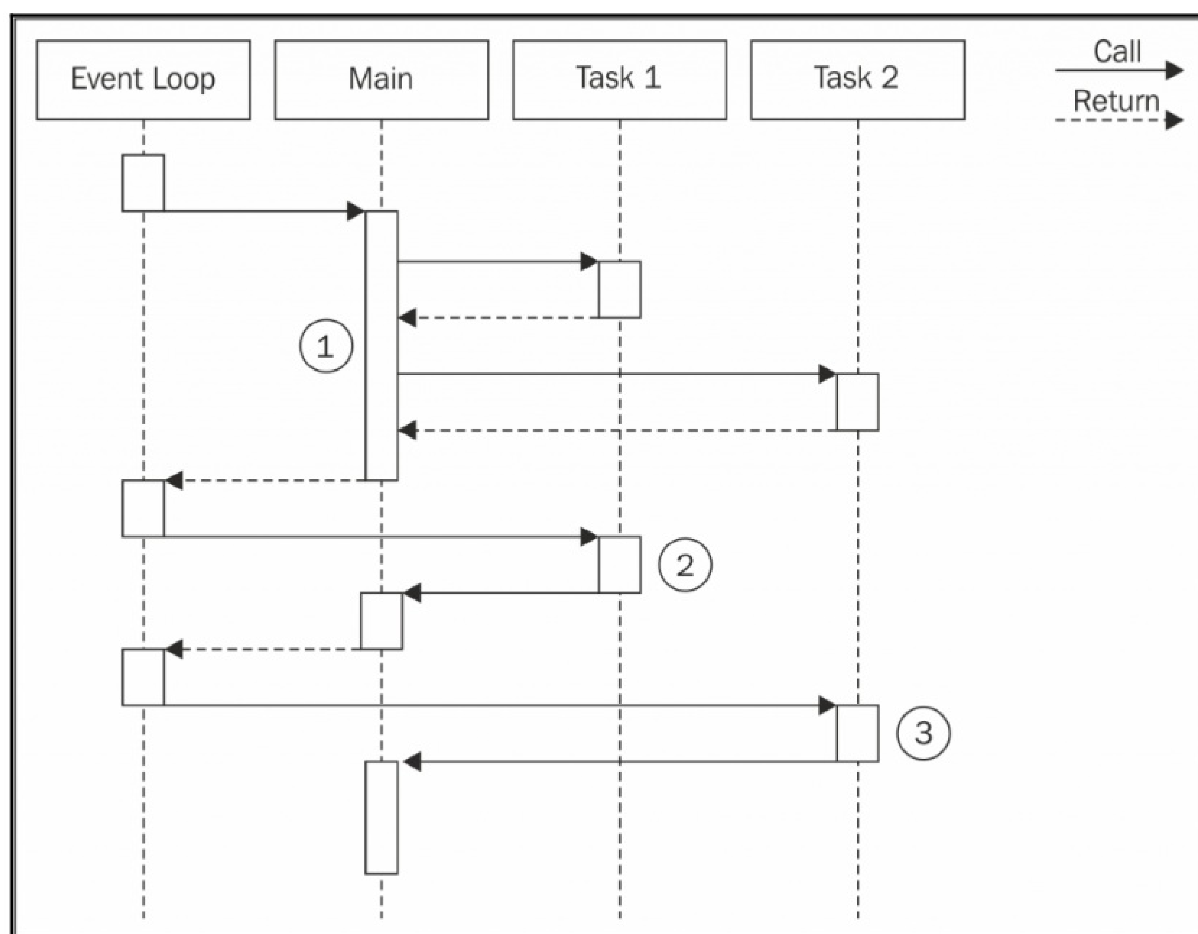
在某些情况下，一组异步任务的执行顺序并不重要，我们只需要在所有这些运行的任务完成时通知我们。使用并行执行流更好地处理这种情况，如下图所示：



如果我们认为 `Node.js` 是单线程的话，这可能听起来很奇怪，但是如果我们记住我们在第一章中讨论过的内容，我们意识到即使我们只有一个线程，我们仍然可以实现并发，由于 `Node.js` 的非阻塞性质。实际上，在这种情况下，并行字不正确地使用，因为这并不意味着任务同时运行，而是它们的执行由底层的非阻塞 `API` 执行，并由事件循环进行交织。

我们知道，当一个任务允许事件循环执行另一个任务时，或者是说一个任务允许控制回到事件循环。这种工作流的名称为并发，但为了简单起见，我们仍然会使用并行。

下图显示了两个异步任务可以在 `Node.js` 程序中并行运行：



通过上图，我们有一个 `Main` 函数执行两个异步任务：

1. `Main` 函数触发 `Task 1` 和 `Task 2` 的执行。由于这些触发异步操作，这两个函数会立即返回，并将控制权返还给主函数，之后等到事件循环完成再通知主线程。
2. 当 `Task 1` 的异步操作完成时，事件循环给与其线程控制权。当 `Task 1` 同步操作完成时，它通知 `Main` 函数。
3. 当 `Task 2` 的异步操作完成时，事件循环给与其线程控制权。当 `Task 2` 同步操作完成时，它再次通知 `Main` 函数。在这一点上， `Main` 函数知晓 `Task 1` 和 `Task 2` 都已经执行完毕，所以它可以继续执行其后操作或将操作的结果返回给另一个回调函数。

简而言之，这意味着在 `Node.js` 中，我们只能执行并行异步操作，因为它们的并发性由非阻塞 `API` 在内部处理。在 `Node.js` 中，同步阻塞操作不能同时运行，除非它们的执行与异步操作交错，或者通过 `setTimeout()` 或 `setImmediate()` 延迟。我们将在第九章中更详细地看到这一点。

Web爬虫版本3

上边的 `Web爬虫` 在并行异步操作上似乎也算表现得很完美。到目前为止，应用程序正在递归地执行链接页面的下载。但性能不是最佳的，想要提升这个应用的性能很容易。

要做到这一点，我们只需要修改 `spiderLinks()` 函数，确保 `spider()` 任务只执行一次，当所有任务都执行完毕后，调用最后的回调，所以我们对 `spiderLinks()` 做如下修改：

```
function spiderLinks(currentUrl, body, nesting, callback) {
  if (nesting === 0) {
    return process.nextTick(callback);
  }
  const links = utilities.getPageLinks(currentUrl, body);
  if (links.length === 0) {
    return process.nextTick(callback);
  }
  let completed = 0,
    hasErrors = false;

  function done(err) {
    if (err) {
      hasErrors = true;
      return callback(err);
    }
    if (++completed === links.length && !hasErrors) {
      return callback();
    }
  }
  links.forEach(link => {
    spider(link, nesting - 1, done);
  });
}
```

上述代码有何变化？，现在 `spider()` 函数的任务全部同步启动。可以通过简单地遍历链接数组和启动每个任务，我们不必等待前一个任务完成再进行下一个任务：

```
links.forEach(link => {
  spider(link, nesting - 1, done);
});
```

然后，使我们的应用程序知晓所有任务完成的方法是为 `spider()` 函数提供一个特殊的回调函数，我们称之为 `done()` 。当爬虫任务完成时， `done()` 函数设定一个计数器。当完成的下载次数达到链接数组的大小时，调用最终回调：

```
function done(err) {
  if (err) {
    hasErrors = true;
    return callback(err);
  }
  if (++completed === links.length && !hasErrors) {
    callback();
  }
}
```

通过上述变化，如果我们现在试图对网页运行我们的爬虫，我们将注意到整个过程的速度有很大的改进，因为每次下载都是并行执行的，而不必等待之前的链接被处理。

模式

此外，对于并行执行流程，我们可以提取我们方案，以便适应于不同的情况提高代码的可复用性。我们可以使用以下代码来表示模式的通用版本：

```
const tasks = [ /* ... */ ];
let completed = 0;
tasks.forEach(task => {
  task(() => {
    if (++completed === tasks.length) {
      finish();
    }
  });
});

function finish() {
  // 所有任务执行完成后调用
}
```

通过小的修改，我们可以调整模式，将每个任务的结果累积到一个 `list` 中，以便过滤或映射数组的元素，或者一旦完成了一个或一定数量的任务即可调用 `finish()` 回调。

> 注意：如果是没有限制的情况下，并行执行的一组异步任务，然后等待所有异步任务完成后执行回调这种方式，其方法是计算它们的执行完成的数目。

用并发任务修复竞争条件

当使用 `阻塞I/O` 与多线程组合的方式时，并行运行一组任务可能会导致一些问题。但是，我们刚刚看到，在 `Node.js` 中却不一样，并行运行多个异步任务实际上在资源方面消耗较低。这是 `Node.js` 最重要的优点之一，因此在 `Node.js` 中并行化成为一种常见的做法，而且这并是多么复杂的技术。

`Node.js` 的并发模型的另一个重要特征是我们处理任务同步和竞争条件的方式。在多线程编程中，这通常使用诸如锁，互斥条件，信号量和观察器之类的构造来实现，这些是多线程语言并行化的最复杂的方面之一，对性能也有很大的影响。在 `Node.js` 中，我们通常不需要一个花哨的同步机制，因为所有运行在单个线程上！但是，这并不意味着我们没有竞争条件。相反，他们可以相当普遍。问题的根源在于异步操作的调用与其结果通知之间的延迟。举一个具体的例子，我们可以再次参考我们的 `Web爬虫` 应用程序，特别是我们创建的最后一个版本，其实际上包含一个竞争条件。

问题在于在开始下载相应的 `URL` 的文档之前，检查文件是否已经存在的 `spider()` 函数：

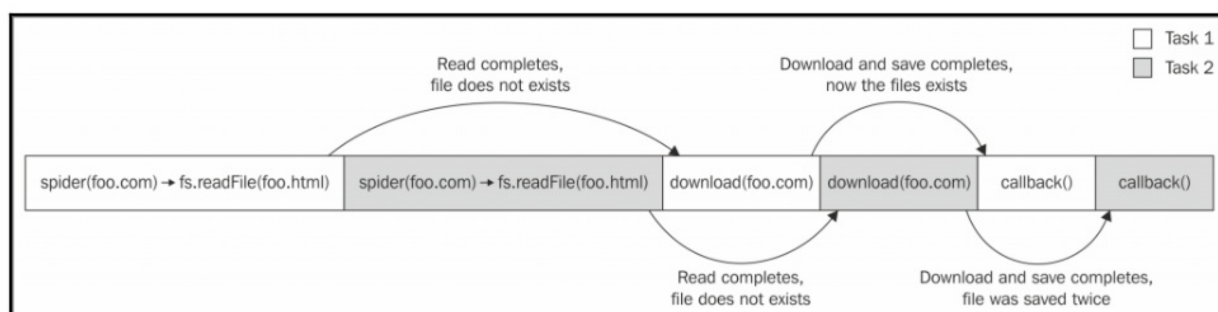
```
function spider(url, nesting, callback) {
  if(spidering.has(url)) {
    return process.nextTick(callback);
  }
  spidering.set(url, true);

  const filename = utilities.urlToFilename(url);
  fs.readFile(filename, 'utf8', function(err, body) {
    if(err) {
      if(err.code !== 'ENOENT') {
        return callback(err);
      }

      return download(url, filename, function(err, body) {
        if(err) {
          return callback(err);
        }
        spiderLinks(url, body, nesting, callback);
      });
    }

    spiderLinks(url, body, nesting, callback);
  });
}
```

现在的问题是，在同一个 `URL` 上操作的两个爬虫任务可能会在两个任务之一完成下载并创建一个文件，导致第二个任务开始下载之前，在同一个文件上调用 `fs.readFile()` 的结果不对，致使下载两次。这种情况如下图所示：



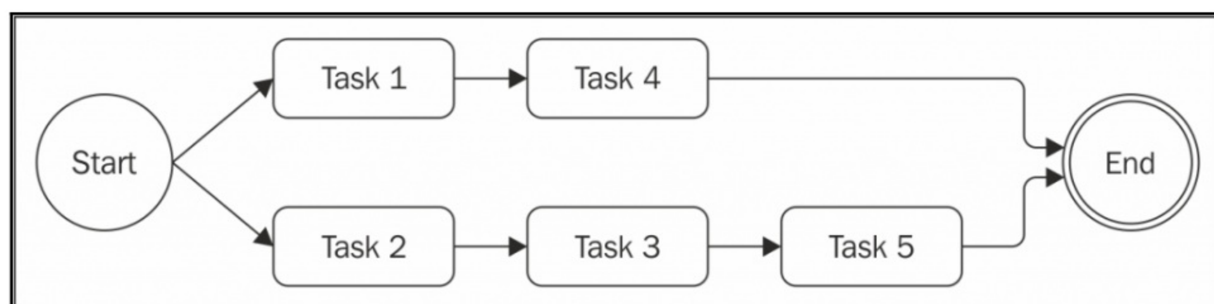
上图显示了 `Task 1` 和 `Task 2` 如何在 `Node.js` 的单个线程中交错执行，以及异步操作如何实际引入竞争条件。在我们的情况下，两个爬虫任务最终会下载相同的文件。 我们如何解决这个问题？答案比我们想象的要简单得多。实际上，我们所需要的只是一个变量（互斥变量），可以相互排除运行在同一个 `URL` 上的多个 `spider()` 任务。这可以通过以下代码来实现：

```
const spidering = new Map();

function spider(url, nesting, callback) {
  if (spidering.has(url)) {
    return process.nextTick(callback);
  }
  spidering.set(url, true);
  // ...
}
```

并行执行频率限制

通常，如果不控制并行任务频率，并行任务就会导致过载。想象一下，有数千个文件要读取，访问的 `URL` 或数据库查询并行运行。在这种情况下，常见的问题是系统资源不足，例如，当尝试一次打开太多文件时，利用可用于应用程序的所有文件描述符。在 `Web应用程序` 中，它还可能会创建一个利用拒绝服务（ `DoS` ）攻击的漏洞。在所有这种情况下，最好限制同时运行的任务数量。这样，我们可以为服务器的负载增加一些可预测性，并确保我们的应用程序不会耗尽资源。下图描述了一个情况，我们将五个任务并行运行并发限制为两段：



从上图可以清楚我们的算法如何工作：

1. 我们可以执行尽可能多的任务，而不超过并发限制。
2. 每当任务完成时，我们再执行一个或多个任务，同时确保任务数量达不到限制。

并发限制

我们现在提出一种模式，以有限的并发性并行执行一组给定的任务：

```
const tasks = ...
let concurrency = 2, running = 0, completed = 0, index = 0;

function next() {
  while (running < concurrency && index < tasks.length) {
    task = tasks[index++];
    task(() => {
      if (completed === tasks.length) {
        return finish();
      }
      completed++, running--;
      next();
    });
    running++;
  }
}
next();

function finish() {
  // 所有任务执行完成
}
```

该算法可以被认为是顺序执行和并行执行之间的混合。事实上，我们可能会注意到我们之前介绍的两种模式的相似之处：

1. 我们有一个迭代器函数，我们称之为 `next()` ，有一个内部循环，并行执行尽可能多的任务，同时保持并发限制。
2. 我们传递给每个任务的回调检查是否完成了列表中的所有任务。如果还有任务要运行，它会调用 `next()` 来执行下一个任务。

全局并发限制

我们的 `Web爬虫` 应用程序非常适合应用我们所学到的限制一组任务的并发性。事实上，为了避免同时爬上数千个链接的情况，我们可以通过在并发下载数量上增加一些措施来限制并发量。

> 0.11之前的Node.js版本已经将每个主机的并发HTTP连接数限制为5.然而，这可以改变以适应我们的需要。请查看官方文档http://nodejs.org/docs/v0.10.0/api/http.html#http_agent_m axsockets中的更多内容。从Node.js 0.11开始，并发连接数没有默认限制。

我们可以将我们刚刚学到的模式应用到我们的 `spiderLinks()` 函数，但是我们将获得的只是限制一个页面中的一组链接的并发性。如果我们选择了并发量为2，我们最多可以为每个页面并行下载两个链接。然而，由于我们可以一次下载多个链接，因此每个页面都会产生另外两个下载，这样递归下去，其实也没有完全做到并发量的限制。

使用队列

我们真正想要的是限制我们可以并行运行的全局下载操作数量。我们可以略微修改之前展示的模式，但是我们宁愿把它作为一个练习，因为我们想借此机会引入另一个机制，它利用队列来限制多个任务的并发性。让我们看看这是如何工作的。

我们现在要实现一个名为 `TaskQueue` 类，它将队列与我们之前提到的算法相结合。我们创建一个名为 `taskQueue.js` 的新模块：

```
class TaskQueue {
  constructor(concurrency) {
    this.concurrency = concurrency;
    this.running = 0;
    this.queue = [];
  }
  pushTask(task) {
    this.queue.push(task);
    this.next();
  }
  next() {
    while (this.running < this.concurrency && this.queue.length)
 {
      const task = this.queue.shift();
      task(() => {
        this.running--;
        this.next();
      });
      this.running++;
    }
  }
};
```

上述类的构造函数只作为输入的并发限制，但除此之外，它初始化运行和队列的变量。前一个变量是用于跟踪所有正在运行的任务的计数器，而后者是将用作队列以存储待处理任务的数组。

`pushTask()` 方法简单地将新任务添加到队列中，然后通过调用 `this.next()` 来引导任务的执行。

`next()` 方法从队列中生成一组任务，确保它不超过并发限制。

我们可能会注意到，这种方法与限制我们前面提到的并发性的模式有一些相似之处。它基本上从队列开始尽可能多的任务，而不超过并发限制。当每个任务完成时，它会更新运行任务的计数，然后再次调用 `next()` 来启动另一轮任务。`TaskQueue` 类的有趣属性是它允许我们动态地将新的项目添加到队列中。另一个优点是，现在我们有一个中央实体负责限制我们任务的并发性，这可以在函数执行的所有实例中共享。在我们的例子中，它是 `spider()` 函数，我们将在稍后看到。

Web爬虫版本4

现在我们有一个通用的队列来执行有限的并行流程中的任务，我们可以在我们的 `Web爬虫` 应用程序中直接使用它。我们首先加载新的依赖关系并通过将并发限制设置为2来创建 `TaskQueue` 类的新实例：

```
const TaskQueue = require('./taskQueue');
const downloadQueue = new TaskQueue(2);
```

接下来，我们使用新创建的 `downloadQueue` 更新 `spiderLinks()` 函数：

```
function spiderLinks(currentUrl, body, nesting, callback) {
  if (nesting === 0) {
    return process.nextTick(callback);
  }
  const links = utilities.getPageLinks(currentUrl, body);
  if (links.length === 0) {
    return process.nextTick(callback);
  }
  let completed = 0,
    hasErrors = false;
  links.forEach(link => {
    downloadQueue.pushTask(done => {
      spider(link, nesting - 1, err => {
        if (err) {
          hasErrors = true;
          return callback(err);
        }
        if (++completed === links.length && !hasErrors) {
          callback();
        }
        done();
      });
    });
  });
}
```

这个函数的这种新的实现是非常容易的，它与这本章前面提到的无限并行执行的算法非常相似。这是因为我们将并发控制委托给 `TaskQueue` 对象，我们唯一要做的就是检查所有任务是否完成。看上述代码中如何定义我们的任务：

- 我们通过提供自定义回调来运行 `spider()` 函数。
- 在回调中，我们检查与 `spiderLinks()` 函数执行相关的所有任务是否完成。当这个条件为真时，我们调用spiderLinks（）函数的最后回调。
- 在我们的任务结束时，我们调用了 `done()` 回调，以便队列可以继续执行。

在我们进行这些小的变化之后，我们现在可以尝试再次运行 `Web爬虫` 应用程序。这一次，我们应该注意到，同时不会有两个以上的下载。

async库

如果我们到目前为止我们分析的每一个控制流程模式看一下，我们可以看到它们可以用作构建可重用和更通用的解决方案的基础。例如，我们可以将无限制的并行执行算法包装到一个接受任务列表的函数中，并行运行它们，并且当它们都完成时调用给定的回调函数。将控制流算法转化为可重用功能的这种方式可以导致更具声明性和表达性的方式来定义异步控制流，这正是async所做的。 `async` 库是一个非常流行的解决方案，在 `Node.js` 和 `JavaScript` 中来说，用于处理异步代码。它提供了一组功能，可以大大简化不同配置中一组任务的执行，并为异步处理集合提供了有用的帮助。即使有其他几个具有相似目标的库，由于它的受欢迎程度，因此 `async` 是 `Node.js` 中的一个事实上的标准。

顺序执行

`async` 库可以在实现复杂的异步控制流程时大大帮助我们，但是一个难题就是选择正确的库来解决问题。例如，对于顺序执行，有大约20个不同的函数可供选择，包括 `eachSeries()` ， `mapSeries()` ， `filterSeries()` ， `rejectSeries()` ， `reduce()` ， `reduceRight()` ， `detectSeries()` ， `concatSeries()` ， `series()` ， `whilst()` ， `doWhilst()` ， `until()` ， `doUntil()` ， `forever()` ， `waterfall()` ， `compose()` ， `seq()` ， `applyEachSeries()` ， `iterator()` ，和 `timesSeries()` 。

选择正确的函数是编写更稳固和可读的代码的重要一步，但这也需要一些经验和实践。在我们的例子中，我们将仅介绍其中的一些情况，但它们仍将为理解和有效地使用库的其余部分提供坚实的基础。

下面，通过例子说明 `async` 库如何工作，我们将用于我们的 `Web爬虫` 应用程序。我们直接从版本2开始，按顺序递归地下载所有的链接。

但是，首先我们确保将 `async` 库安装到我们当前的项目中：

```
npm install async
```

然后我们需要从 `spider.js` 模块加载新的依赖项：

```
const async = require('async');
```

已知一组任务的顺序执行

我们先修改 `download()` 函数。如下所示，它依次做了以下三件事：

1. 下载 `URL` 的内容。
2. 创建一个新目录（如果尚不存在）。
3. 将 `URL` 的内容保存到文件中。

`async.series()` 可以实现顺序执行一组任务：

```
async.series(tasks, [callback])
```

`async.series()` 接受一个任务列表和一个在所有任务完成后调用的回调函数作为参数。每个任务只是一个接受回调函数的函数，当任务完成执行时，这个回调函数被调用：

```
function task(callback) {}
```

`async` 的优势是它使用与 `Node.js` 相同的回调约定，它会自动处理错误传播。所以，如果任何一个任务调用它的回调并且产生了一个错误，`async` 将跳过列表中剩余的任务，直接跳转到最后的回调。

考虑到这一点，让我们看看如何通过使用 `async` 来修改上述的 `download()` 函数：

```
function download(url, filename, callback) {
  console.log(`Downloading ${url}`);
  let body;
  async.series([
    callback => {
      request(url, (err, response, resBody) => {
        if (err) {
          return callback(err);
        }
        body = resBody;
        callback();
      });
    },
    mkdirp.bind(null, path.dirname(filename)),
    callback => {
      fs.writeFile(filename, body, callback);
    }
  ], err => {
    if (err) {
      return callback(err);
    }
    console.log(`Downloaded and saved: ${url}`);
    callback(null, body);
  });
}
```

对比起这段代码的回调地狱版本，使用 `async` 方式使我们能够更好地组织我们的异步任务。并且不会嵌套回调，因为我们只需要提供一个的任务列表，通常对于用于每个异步操作，然后异步任务将依次执行：

1. 首先是下载 `URL` 的内容。我们将响应体保存到一个闭包变量（ `body` ）中，以便它可以与其他任务共享。
2. 创建并保存下载的页面的目录。我们通过执行 `mkdirp()` 函数实现，并和创建的目录路径绑定。这样，我们可以节省几行代码并增加其可读性。
3. 最后，我们将下载的 `URL` 的内容写入文件。在这种情况下，我们无法执行部分应用程序（就像我们在第二个任务中所做的那样），因为变量 `body` 只在系列中的下载任务完成后才可用。但是，通过将任务的回调直接传递到 `fs.writeFile()` 函数，我们仍然可以通过利用异步的自动错误管理来保存一些代码行。 4.完成所有任务后，将调用 `async.series()` 的最后回调。在我们的例子中，我们只是做一些错误管理，然后返回 `body` 变量来回调 `download()` 函数。

对于上述情况， `async.series()` 的一个可替代的方法是 `async.waterfall()` ，它仍然按顺序执行任务，但另外还提供每个任务的输出作为下一个输入。在我们的情况下，我们可以使用这个特征来传播 `body` 变量直到序列结束。

顺序迭代

在前面讲了如何按顺序执行一组任务。上面的例子 `async.series()` 来做到这一点。可以使用相同的功能来实现 `Web爬虫版本2` 的 `spiderLinks()` 函数。然而， `async` 为特定的情况提供了一个更合适的 `API` ，遍历一个集合，这个 `API` 是 `async.eachSeries()` 。我们来使用它来重新实现我们的 `spiderLinks()` 函数（版本2，串行下载），如下所示：

```
function spiderLinks(currentUrl, body, nesting, callback) {
  if (nesting === 0) {
    return process.nextTick(callback);
  }
  const links = utilities.getPageLinks(currentUrl, body);
  if (links.length === 0) {
    return process.nextTick(callback);
  }
  async.eachSeries(links, (link, callback) => {
    spider(link, nesting - 1, callback);
  }, callback);
}
```

如果我们将使用 `async` 的上述代码与使用纯 `JavaScript` 模式实现的相同功能的代码进行比较，我们将注意到 `async` 在代码组织和可读性方面给我们带来的巨大优势。

并行执行

`async` 不具有处理并行流的功能，其中可以找到 `each()` ，`map()` ，`filter()` ，`reject()` ，`detect()` ，`some()` ，`ev`，`concat()` ，`parallel()` ，`applyEach()` 和 `times()` 。它们遵循与我们已经看到的用于顺序执行的功能相同的逻辑，区别在于所提供的任务是并行执行的。

为了证明这一点，我们可以尝试应用上述功能之一来实现我们的 `Web爬虫` 应用程序的第三版，即使用无限制的并行流程来执行下载。

如果我们记住我们之前使用的代码来实现 `spiderLinks()` 函数的顺序版本，那么调整它使其并行工作就比较简单：

```
function spiderLinks(currentUrl, body, nesting, callback) {
  // ...
  async.each(links, (link, callback) => {
    spider(link, nesting - 1, callback);
  }, callback);
}
```

这个函数与我们用于顺序下载的功能完全相同，但是使用的是 `async.each()` 而非 `async.eachSeries()` 。这清楚地表明了使用库（例如 `async` ）抽象异步流的功能。代码不再绑定到特定的执行流程了，没有专门为此写的代码。大多数只是应用逻辑。

限制并行执行

如果你想知道 `async` 还可以用来限制并行任务的并发性，答案是肯定的。我们有一些我们可以使用的函数，即 `eachLimit()` ，`mapLimit()` ，`parallelLimit()` ，`queue()` 和 `cargo()` 。

我们试图利用其中的一个来实现 `Web爬虫` 应用程序的第4版，以有限的并发性并行执行链接的下载。幸运的是， `async` 有 `async.queue()` ，它的工作方式与本章前面创建的 `TaskQueue` 类似。 `async.queue()` 函数创建一个新的队列，它使用一个 `worker()` 函数来执行一组具有指定并发限制的任务：

```
const q = async.queue(worker, concurrency);
```

`worker()` 函数作为输入接收要运行的任务和一个回调函数作为参数，当任务完成时执行回调：

```
function worker(task, callback);
```

我们应该注意到在这个例子中 `task` 可以是任何类型，而不仅仅只能是函数。实际上， `worker` 有责任以最适当的方式处理任务。新建任务，可以通过 `q.push(task, callback)` 将任务添加到队列中。一个任务处理完后，关联一个任务的回调函数必须被 `worker` 调用。

现在，我们再次修改我们的代码实现一个全面并行的有并发限制的执行流，利用 `async.queue()` ,首先，我们需要创建一个队列:

```
const downloadQueue = async.queue((taskData, callback) => {
  spider(taskData.link, taskData.nesting - 1, callback);
}, 2);
```

代码很简单。我们正在创建一个并发限制为2的新队列，让一个工作人员只需使用与任务关联的数据调用我们的 `spider()` 函数。接下来，我们实现 `spiderLinks()` 函数：

```
function spiderLinks(currentUrl, body, nesting, callback) {
  if (nesting === 0) {
    return process.nextTick(callback);
  }
  const links = utilities.getPageLinks(currentUrl, body);
  if (links.length === 0) {
    return process.nextTick(callback);
  }
  const completed = 0,
    hasErrors = false;
  links.forEach(function(link) {
    const taskData = {
      link: link,
      nesting: nesting
    };
    downloadQueue.push(taskData, err => {
      if (err) {
        hasErrors = true;
        return callback(err);
      }
      if (++completed === links.length && !hasErrors) {
        callback();
      }
    });
  });
}
```

前面的代码应该看起来非常熟悉，因为它几乎和使用 `TaskQueue` 对象来实现相同流程的代码相同。此外，在这种情况下，要分析的重要部分是将新任务推入队列的位置。在这一点上，我们确保我们传递一个回调，使我们能够检查当前页面的所有下载任务是否完成，并最终调用最终回调。

幸亏有 `async.queue()` ，我们可以轻松地复制我们的 `TaskQueue` 对象的功能，再次证明了通过 `async` ，我们可以避免从头开始编写异步控制流模式，减少我们的工作量，代码量更加简洁。

总结

在本章开始的时候，我们说 `Node.js` 的编程可能很难因为它的异步性，特别是对于以前在其他平台上开发的人而言。然而，在本章中，我们展示了异步 `API` 如何可以从简单原生 `JavaScript` 开始，从而为我们分析更复杂的技术奠定了基础。然后我们看到，除了为每一种口味提供编程风格，我们所掌握的工具确实是多样化的，并为我们大部分的问题提供了很好的解决方案。例如，我们可以选择 `async` 库来简化最常见的流程。

还有更为先进的技术，如 `Promise` 和 `Generator` 函数，这将是下一章的重点。当了解所有这些技术时，能够根据需求选择最佳解决方案，或者在同一个项目中使用多种技术。

Asynchronous Control Flow Patterns with ES2015 and Beyond

在上一章中，我们学习了如何使用回调处理异步代码，以及如何解决如回调地狱代码等异步问题。回调是 `JavaScript` 和 `Node.js` 中的异步编程的基础，但是现在，其他替代方案已经出现。这些替代方案更复杂，以便能够以更方便的方式处理异步代码。

在本章中，我们将探讨一些代表性的替代方案， `Promise` 和 `Generator` 。以及 `async await` ，这是一种创新的语法，可在高版本的 `JavaScript` 中提供，其也作为 `ECMAScript 2017` 发行版的一部分。

我们将看到这些替代方案如何简化处理异步控制流的方式。最后，我们将比较所有这些方法，以了解所有这些方法的所有优点和缺点，并能够明智地选择最适合我们下一个 `Node.js` 项目要求的方法。

Promise

我们在前面的章节中提到， `CPS风格` 不是编写异步代码的唯一方法。事实上， `JavaScript` 生态系统为传统的回调模式提供了有趣的替代方案。最着名的选择之一是 `Promise` ，特别是现在它是 `ECMAScript 2015` 的一部分，并且现在可以在 `Node.js` 中可用。

什么是**Promise**？

`Promise` 是一种抽象的对象，我们通常允许函数返回一个名为 `Promise` 的对象，它表示异步操作的最终结果。通常情况下，我们说当异步操作尚未完成时，我们说 `Promise` 对象处于 `pending` 状态，当操作成功完成时，我们说 `Promise` 对象处于 `resolve` 状态，当操作错误终止时，我们说 `Promise` 对象处于 `reject` 状态。一旦 `Promise` 处于 `resolve` 或 `reject` ，我们认为当前异步操作结束。

为了接收到异步操作的正确结果或错误捕获，我们可以使用 `Promise` 的 `then` 方法：

```
promise.then([onFulfilled], [onRejected])
```

在前面的代码中， `onFulfilled()` 是一个函数，最终会收到 `Promise` 的正确结果，而 `onRejected()` 是另一个函数，它将接收产生异常的原因（如果有的话）。两个参数都是可选的。

要了解 `Promise` 如何转换我们的代码，让我们考虑以下几点：

```
asyncOperation(arg, (err, result) => {
  if (err) {
    // 错误处理
  }
  // 正常结果处理
});
```

`Promise` 允许我们将这个典型的 `CPS` 代码转换成更好的结构化和更优雅的代码，如下所示：

```
asyncOperation(arg)
  .then(result => {
    // 错误处理
  }, err => {
    // 正常结果处理
  });
```

`then()` 方法的一个关键特征是它同步地返回另一个 `Promise` 对象。如果 `onFulfilled()` 或 `onRejected()` 函数中的任何一个函数返回 `x`，则 `then()` 方法返回的 `Promise` 对象将如下所示：

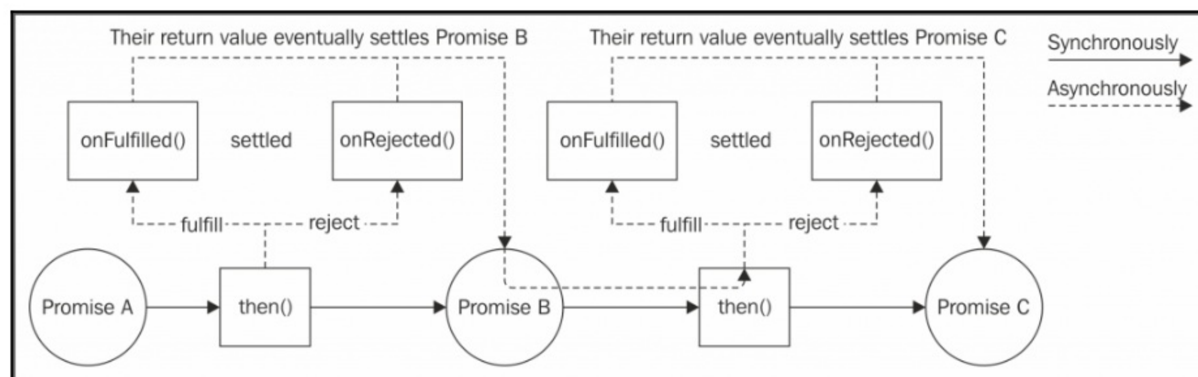
- 如果 `x` 是一个值，则这个 `Promise` 对象会正确处理(`resolve`) `x`
- 如果 `x` 是一个 `Promise` 对象或 `thenable`，则会正确处理(`resolve`) `x`
- 如果 `x` 是一个异常，则会捕获异常(`reject`) `x`

> 注：thenable是一个具有then方法的类似于Promise的对象(Promise-like)。

这个特点使我们能够链式构建 `Promise`，允许轻松排列组合我们的异步操作。另外，如果我们没有指定一个 `onFulfilled()` 或 `onRejected()` 处理程序，则正确结果或异常捕获将自动转发到 `Promise` 链的下一个 `Promise`。例如，这允许我们在整个链中自动传播错误，直到被 `onRejected()` 处理程序捕获。随着 `Promise` 链，任务的顺序执行突然变成简单多了：

```
asyncOperation(arg)
  .then(result1 => {
    // 返回另一个Promise
    return asyncOperation(arg2);
  })
  .then(result2 => {
    // 返回一个值
    return 'done';
  })
  .then(undefined, err => {
    // 捕获Promise链中的异常
  });
```

下图展示了链式 `Promise` 如何工作：



`Promise` 的另一个重要特性是 `onFulfilled()` 和 `onRejected()` 函数是异步调用的，如同上述的例子，在最后那个 `then` 函数 `resolve` 一个同步的 `Promise` ，它也是同步的。这种模式避免了 `Zalgo` （参见 `Chapter2-Node.js Essential Patterns` ），使我们的异步代码更加一致和稳健。

如果在 `onFulfilled()` 或 `onRejected()` 处理程序中抛出异常（使用 `throw` 语句），则 `then()` 方法返回的 `Promise` 将被自动地 `reject` ，抛出异常作为 `reject` 的原因。这相对于 `CPS` 来说是一个巨大的优势，因为它意味着有了 `Promise` ，异常将在整个链中自动传播，并且 `throw` 语句终于可以使用。

在以前，许多不同的库实现了 `Promise` ，大多数时候它们之间不兼容，这意味着不可能在使用不同 `Promise` 库的 `thenable` 链式传播错误。

`JavaScript` 社区非常努力地解决了这个限制，这些努力导致了 `Promises / A +` 规范的创建。该规范详细描述了 `then` 方法的行为，提供了一个可互兼容的基础，这使得来自不同库的 `Promise` 对象能够彼此兼容，开箱即用。

有关 `Promises / A +` 规范的详细说明，可以参考Promises / A + 官方网站。

Promise / A + 的实施

在 `JavaScript` 中以及 `Node.js` 中，有几个实现 `Promises / A +` 规范的库。以下是最受欢迎的：

- Bluebird
- Q
- RSVP
- Vow
- When.js
- ES2015 promises

真正区别他们的是在 `Promises / A +` 标准之上提供的额外功能。正如我们上述所说的那样，该标准定义了 `then()` 方法和 `Promise` 解析过程的行为，但它没有指定其他功能，例如，如何从基于回调的异步函数创建 `Promise` 。

在我们的示例中，我们将使用由 `ES2015` 的 `Promise` ，因为 `Promise` 对象自 `Node.js 4` 后即可使用，而不需要任何库来实现。

作为参考，以下是 `ES2015` 的 `Promise` 提供的API：

`constructor` (`new Promise（function（resolve, reject）{}）`)：创建了一个新的 `Promise` ，它基于作为传递两个类型为函数的参数来决定 `resolve` 或 `reject` 。构造函数的参数解释如下：

- `resolve(obj)` ： `resolve` 一个 `Promise` ，并带上一个参数 `obj` ，如果 `obj` 是一个值，这个值就是传递的异步操作成功的结果。如果 `obj` 是一个 `Promise` 或一个 `thenable` ，则会进行正确处理。
- `reject(err)` ： `reject` 一个 `Promise` ，并带上一个参数 `err` 。它是 `Error` 对象的一个实例。

Promise对象的静态方法

- `Promise.resolve(obj)` ： 将会创建一个 `resolve` 的 `Promise` 实例
- `Promise.reject(err)` ： 将会创建一个 `reject` 的 `Promise` 实例
- `Promise.all(iterable)` ：返回一个新的 `Promise` 实例，并且在 `iterable` 中所有 `Promise` 状态为 `reject` 时，返回的 `Promise` 实例的状态会被置为 `reject` ，如果 `iterable` 中至少有一个 `Promise` 状态为 `reject` 时，返回的 `Promise` 实例状态也会被置为 `reject` ，并且 `reject` 的原因是第一个被 `reject` 的 `Promise` 对象的 `reject` 原因。
- `Promise.race(iterable)` ：返回一个 `Promise` 实例，当 `iterable` 中任何一个 `Promise` 被 `resolve` 或被 `reject` 时， 返回的 `Promise` 实例以同样的原因 `resolve` 或 `reject` 。

Promise实例方法

- `Promise.then(onFulfilled, onRejected)` ：这是 `Promise` 的基本方法。它的行为与我们之前描述的 `Promises / A +` 标准兼容。
- `Promise.catch(onRejected)` ：这只是 `Promise.then(undefined，onRejected)` 的语法糖。

> 值得一提的是，一些Promise实现提供了另一种机制来创建新的Promise，称为deferreds。我们不会在这里描述，因为它不是ES2015标准的一部分，但是如果您想了解更多信息，可以阅读Q文档 (https://github.com/kriskowal/q#using-deferreds) 或When.js文档 (https://github.com/cujojs/when/wiki/Deferred) 。

Promisifying一个Node.js回调风格的函数

在 `JavaScript` 中，并不是所有的异步函数和库都支持开箱即用的 `Promise` 。大多数情况下，我们必须将一个典型的基于回调的函数转换成一个返回 `Promise` 的函数，这个过程也被称为 `promisification` 。

幸运的是， `Node.js` 中使用的回调约定允许我们创建一个可重用的函数，我们通过使用 `Promise` 对象的构造函数来简化任何 `Node.js` 风格的 `API` 。让我们创建一个名为 `promisify()` 的新函数，并将其包含到 `utilities.js` 模块中（以便稍后在我们的 `Web爬虫应用程序` 中使用它）：

```
module.exports.promisify = function(callbackBasedApi) {
  return function promisified() {
    const args = [].slice.call(arguments);
    return new Promise((resolve, reject) => {
      args.push((err, result) => {
        if (err) {
          return reject(err);
        }
        if (arguments.length <= 2) {
          resolve(result);
        } else {
          resolve([].slice.call(arguments, 1));
        }
      });
      callbackBasedApi.apply(null, args);
    });
  }
};
```

前面的函数返回另一个名为 `promisified()` 的函数，它表示输入中给出的 `callbackBasedApi` 的 `promisified` 版本。以下展示它是如何工作的：

1. `promisified()` 函数使用 `Promise` 构造函数创建一个新的 `Promise` 对象，并立即将其返回给调用者。
2. 在传递给 `Promise` 构造函数的函数中，我们确保传递给 `callbackBasedApi` ，这是一个特殊的回调函数。由于我们知道回调总是最后调用的，我们只需将回调函数附加到提供给 `promisified()` 函数的参数列表里（ `args` ）。
3. 在特殊的回调中，如果我们收到错误，我们立即 `reject` 这个 `Promise` 。
4. 如果没有收到错误，我们使用一个值或一个数组值来 `resolve` 这个 `Promise` ，具体取决于传递给回调的结果数量。
5. 最后，我们只需使用我们构建的参数列表调用 `callbackBasedApi` 。

> 大部分的Promise已经提供了一个开箱即用的接口来将一个Node.js风格的API转换成一个返回Promise的API。例如，Q有Q.denodeify()和Q.nbind()，Bluebird有Promise.promisify()，而When.js有node.lift()。

顺序执行

在一些必要的理论之后，我们现在准备将我们的 `Web爬虫应用程序` 转换为使用 `Promise` 的形式。让我们直接从版本2开始，直接下载一个Web网页的链接。

在 `spider.js` 模块中，第一步是加载我们的 `Promise` 实现（我们稍后会使用它）和 `Promisifying` 我们打算使用的基于回调的函数：

```
const utilities = require('./utilities');
const request = utilities.promisify(require('request'));
const mkdirp = utilities.promisify(require('mkdirp'));
const fs = require('fs');
const readFile = utilities.promisify(fs.readFile);
const writeFile = utilities.promisify(fs.writeFile);
```

现在，我们开始更改我们的 `download` 函数：

```
function download(url, filename) {
  console.log(`Downloading ${url}`);
  let body;
  return request(url)
    .then(response => {
      body = response.body;
      return mkdirp(path.dirname(filename));
    })
    .then(() => writeFile(filename, body))
    .then(() => {
      console.log(`Downloaded and saved: ${url}`);
      return body;
    });
}
```

这里要注意的到的最重要的是我们也为 `readFile()` 返回的 `Promise` 注册一个 `onRejected()` 函数，用来处理一个网页没有被下载的情况(或文件不存在)。还有，看我们如何使用 `throw` 来传递 `onRejected()` 函数中的错误的。

既然我们已经更改我们的 `spider()` 函数，我们这么修改它的调用方式：

```
spider(process.argv[2], 1)
  .then(() => console.log('Download complete'))
  .catch(err => console.log(err));
```

注意我们是如何第一次使用 `Promise` 的语法糖 `catch` 来处理源自 `spider()` 函数的任何错误情况。如果我们再看看迄今为止我们所写的所有代码，那么我们会惊喜的发现，我们没有包含任何错误传播逻辑，因为我们在使用回调函数时会被迫做这样的事情。这显然是一个巨大的优势，因为它极大地减少了我们代码中的样板文件以及丢失任何异步错误的机会。

现在，完成我们唯一缺失的 `Web爬虫应用程序` 的第二版的 `spiderLinks()` 函数，我们将在稍后实现它。

顺序迭代

到目前为止， `Web爬虫应用程序` 代码库主要是对 `Promise` 是什么以及如何使用的概述，展示了使用 `Promise` 实现顺序执行流程的简单性和优雅性。但是，我们现在考虑的代码只涉及到一组已知的异步操作的执行。所以，完成我们对顺序执行流程的探索的缺失部分是看我们如何使用 `Promise` 来实现迭代。同样，网络蜘蛛第二版的 `spiderLinks()` 函数也是一个很好的例子。

让我们添加缺少的这一块：

```
function spiderLinks(currentUrl, body, nesting) {
  let promise = Promise.resolve();
  if (nesting === 0) {
    return promise;
  }
  const links = utilities.getPageLinks(currentUrl, body);
  links.forEach(link => {
    promise = promise.then(() => spider(link, nesting - 1));
  });
  return promise;
}
```

为了异步迭代一个网页的全部链接，我们必须动态创建一个 `Promise` 的迭代链。

1. 首先，我们定义一个空的 `Promise` ， `resolve` 为 `undefined` 。这个 `Promise` 只是用来作为 `Promise` 的迭代链的起始点。
2. 然后，我们通过在循环中调用链中前一个 `Promise` 的 `then()` 方法获得的新的 `Promise` 来更新 `Promise` 变量。这就是我们使用 `Promise` 的异步迭代模式。

这样，循环的结束， `promise` 变量会包含循环中最后一个 `then()` 返回的 `Promise` 对象，所以它只有当 `Promise` 的迭代链中全部 `Promise` 对象被 `resolve` 后才能被 `resolve` 。

> 注：在最后调用了这个then方法来resolve这个Promise对象

通过这个，我们已使用 `Promise` 对象重写了我们的 `Web爬虫应用程序` 。我们现在应该可以运行它了。

顺序迭代模式

为了总结这个顺序执行的部分，让我们提取一个模式来依次遍历一组 `Promise` ：

```
let tasks = [ /* ... */ ]
let promise = Promise.resolve();
tasks.forEach(task => {
  promise = promise.then(() => {
    return task();
  });
});
promise.then(() => {
  // 所有任务都完成
});
```

使用 `reduce()` 方法来替代 `forEach()` 方法，允许我们写出更为简洁的代码：

```
let tasks = [ /* ... */ ]
let promise = tasks.reduce((prev, task) => {
  return prev.then(() => {
    return task();
  });
}, Promise.resolve());

promise.then(() => {
  //All tasks completed
});
```

与往常一样，通过对这种模式的简单调整，我们可以将所有任务的结果收集到一个数组中，我们可以实现一个 `mapping` 算法，或者构建一个 `filter` 等等。

> 上述这个模式使用循环动态地建立一个链式的Promise。

并行执行

另一个适合用 `Promise` 的执行流程是并行执行流程。实际上，我们需要做的就是使用内置的 `Promise.all()` 。这个方法创造了另一个 `Promise` 对象，只有在输入中的所有 `Promise` 都 `resolve` 时才能 `resolve` 。这是一个并行执行，因为在其参数 `Promise` 对象的之间没有执行顺序可言。

为了演示这一点，我们来看我们的 `Web爬虫应用程序` 的第三版，它将页面中的所有链接并行下载。让我们再次使用 `Promise` 更新 `spiderLinks()` 函数来实现并行流程：

```
function spiderLinks(currentUrl, body, nesting) {
  if (nesting === 0) {
    return Promise.resolve();
  }
  const links = utilities.getPageLinks(currentUrl, body);
  const promises = links.map(link => spider(link, nesting - 1));
  return Promise.all(promises);
}
```

这里的模式在 `elements.map()` 迭代中产生一个数组，存放所有异步任务，之后便于同时启动 `spider()` 任务。这一次，在循环中，我们不等待以前的下载完成，然后开始一个新的下载任务：所有的下载任务在一个循环中一个接一个地开始。之后，我们利用 `Promise.all()` 方法，它返回一个新的 `Promise` 对象，当数组中的所有 `Promise` 对象都被 `resolve` 时，这个 `Promise` 对象将被 `resolve` 。换句话说，所有的下载任务完成，这正是我们想要的。

限制并行执行

不幸的是， `ES2015` 的 `Promise API` 并没有提供一种原生的方式来限制并发任务的数量，但是我们总是可以依靠我们所学到的有关用普通 `JavaScript` 来限制并发。事实上，我们在 `TaskQueue` 类中实现的模式可以很容易地被调整来支持返回承诺的任务。这很容易通过修改 `next()` 方法来完成：

```
class TaskQueue {
  constructor(concurrency) {
    this.concurrency = concurrency;
    this.running = 0;
    this.queue = [];
  }

  pushTask(task) {
    this.queue.push(task);
    this.next();
  }

  next() {
    while (this.running < this.concurrency && this.queue.length)
 {
      const task = this.queue.shift();
      task().then(() => {
        this.running--;
        this.next();
      });
      this.running++;
    }
  }
}
```

不同于使用一个回调函数来处理任务，我们简单地调用 `Promise` 的 `then()` 。

让我们回到 `spider.js` 模块，并修改它以支持我们的新版本的 `TaskQueue` 类。首先，我们确保定义一个 `TaskQueue` 的新实例：

```
const TaskQueue = require('./taskQueue');
const downloadQueue = new TaskQueue(2);
```

然后，是我们的 `spiderLinks()` 函数。这里的修改也是很简单：

```
function spiderLinks(currentUrl, body, nesting) {
  if (nesting === 0) {
    return Promise.resolve();
  }
  const links = utilities.getPageLinks(currentUrl, body);
  // 我们需要如下代码，用于创建Promise对象
  // 如果没有下列代码，当任务数量为0时，将永远不会resolve
  if (links.length === 0) {
    return Promise.resolve();
  }
  return new Promise((resolve, reject) => {
    let completed = 0;
    let errored = false;
    links.forEach(link => {
      let task = () => {
        return spider(link, nesting - 1)
          .then(() => {
            if (++completed === links.length) {
              resolve();
            }
          })
          .catch(() => {
            if (!errored) {
              errored = true;
              reject();
            }
          });
      };
      downloadQueue.pushTask(task);
    });
  });
}
```

在上述代码中有几点值得我们注意的：

- 首先，我们需要返回使用 `Promise` 构造函数创建的新的 `Promise` 对象。正如我们将看到的，这使我们能够在队列中的所有任务完成时手动 `resolve` 我们的 `Promise` 对象。
- 然后，我们应该看看我们如何定义任务。我们所做的是将一

个 `onFulfilled()` 回调函数的调用添加到由 `spider()` 返回的 `Promise` 对象中，所以我们可以计算完成的下载任务的数量。当完成的下载量与当前页面中链接的数量相同时，我们知道任务已经处理完毕，所以我们可以调用外部 `Promise` 的 `resolve()` 函数。

> Promises / A +规范规定，then()方法的onFulfilled()和onRejected()回调函数只能调用一次（仅调用onFulfilled()和onRejected()）。Promise接口的实现确保即使我们多次手动调用resolve或reject，Promise也仅可以被resolve或reject一次。

现在，使用 `Promise` 的 `Web爬虫应用程序` 的第4版应该已经准备好了。我们可能再次注意到下载任务如何并行运行，并发数量限制为2。

在公有API中暴露回调函数和Promise

正如我们在前面所学到的， `Promise` 可以被用作回调函数的一个很好的替代品。它们使我们的代码更具可读性和易于理解。虽然 `Promise` 带来了许多优点，但也要求开发人员理解许多不易于理解的概念，以便正确和熟练地使用。由于这个原因和其他原因，在某些情况下，比起 `Promise` 来说，很多开发者更偏向于回调函数。

现在让我们想象一下，我们想要构建一个执行异步操作的公共库。我们需要做什么？我们是创建了一个基于回调函数的 `API` 还是一个面向 `Promise` 的 `API` ？还是两者均有？

这是许多知名的库所面临的问题，至少有两种方法值得一提，使我们能够提供一个多功能的 `API` 。

像 `request` ， `redis` 和 `mysql` 这样的库所使用的第一种方法是提供一个简单的基于回调函数的 `API` ，如果需要，开发人员可以选择公开函数。其中一些库提供工具函数来 `Promise` 化异步回调，但开发人员仍然需要以某种方式将暴露的 `API` 转换为能够使用 `Promise` 对象。

第二种方法更透明。它还提供了一个面向回调的 `API` ，但它使回调参数可选。每当回调作为参数传递时，函数将正常运行，在完成时或失败时执行回调。当回调未被传递时，函数将立即返回一个 `Promise` 对象。这种方法有效地结合了回调函数和 `Promise` ，使得开发者可以在调用时选择采用什么接口，而不需要提前进行 `Promise` 化。许多库，如 `mongoose` 和 `sequelize` ，都支持这种方法。

我们来看一个简单的例子。假设我们要实现一个异步执行除法的模块：

```
module.exports = function asyncDivision(dividend, divisor, cb) {
  return new Promise((resolve, reject) => { // [1]
    process.nextTick(() => {
      const result = dividend / divisor;
      if (isNaN(result) || !Number.isFinite(result)) {
        const error = new Error('Invalid operands');
        if (cb) {
          cb(error); // [2]
        }
        return reject(error);
      }
      if (cb) {
        cb(null, result); // [3]
      }
      resolve(result);
    });
  });
};
```

该模块的代码非常简单，但是有一些值得强调的细节：

- 首先，返回使用 `Promise` 的构造函数创建的新承诺。我们在构造函数参数函数内定义全部逻辑。
- 在发生错误的情况下，我们 `reject` 这个 `Promise` ，但如果回调函数在被调用时作为参数传递，我们也执行回调来进行错误传播。
- 在计算结果之后，我们 `resolve` 了这个 `Promise` ，但是如果有回调函数，我们也会将结果传播给回调函数。

我们现在看如何用回调函数和 `Promise` 来使用这个模块：

```
// 回调函数的方式
asyncDivision(10, 2, (error, result) => {
  if (error) {
    return console.error(error);
  }
  console.log(result);
});

// Promise化的调用方式
asyncDivision(22, 11)
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

应该很清楚的是，即将开始使用类似于上述的新模块的开发人员将很容易地选择最适合自己需求的风格，而无需在希望利用 `Promise` 时引入外部 `promisification` 功能。

Generators

`ES2015` 规范引入了另外一种机制，除了其他新功能外，还可以用来简化 `Node.js` 应用程序的异步控制流程。我们正在谈论 `Generator` ，也被称为 `semi-coroutines` 。它们是子程序的一般化，可以有不同的入口点。在一个正常的函数中，实际上我们只能有一个入口点，这个入口点对应着函数本身的调用。 `Generator` 与一般函数类似，但是可以暂停（使用 `yield` 语句），然后在稍后继续执行。在实现迭代器时， `Generator` 特别有用，因为我们已经讨论了如何使用迭代器来实现重要的异步控制流模式，如顺序执行和限制并行执行。

Generators基础

在我们探索使用 `Generator` 来实现异步控制流程之前，学习一些基本概念是很重要的。我们从语法开始吧。可以通过在函数关键字之后附加 `*` （星号）运算符来声明 `Generator` 函数：

```
function* makeGenerator() {
  // body
}
```

在 `makeGenerator()` 函数内部，我们可以使用关键字 `yield` 暂停执行并返回给调用者传递给它的值：

```
function* makeGenerator() {
  yield 'Hello World';
  console.log('Re-entered');
}
```

在前面的代码中， `Generator` 通过 `yield` 一个字符串 `Hello World` 暂停当前函数的执行。当 `Generator` 恢复时，执行将从下列语句开始：

```
console.log('Re-entered');
```

`makeGenerator()` 函数本质上是一个工厂，它在被调用时返回一个新的 `Generator` 对象：

```
const gen = makeGenerator();
```

生成器对象的最重要的方法是 `next()` ，它用于启动/恢复 `Generator` 的执行，并返回如下形式的对象：

```
{
  value: <yielded value>
  done: <true if the execution reached the end>
}
```

这个对象包含 `Generator` `yield` 的值和一个指示 `Generator` 是否已经完成执行的符号。

一个简单的例子

为了演示 `Generator` ，我们来创建一个名为 `fruitGenerator.js` 的新模块：

```
function* fruitGenerator() {
  yield 'apple';
  yield 'orange';
  return 'watermelon';
}
const newFruitGenerator = fruitGenerator();
console.log(newFruitGenerator.next()); // [1]
console.log(newFruitGenerator.next()); // [2]
console.log(newFruitGenerator.next()); // [3]
```

前面的代码将打印下面的输出：

```
{ value: 'apple', done: false }
{ value: 'orange', done: false }
{ value: 'watermelon', done: true }
```

我们可以这么解释上述现象：

- 第一次调用 `newFruitGenerator.next()` 时， `Generator` 函数开始执行，直到达到第一个 `yield` 语句为止，该命令暂停 `Generator` 函数执行，并将值 `apple` 返回给调用者。
- 在第二次调用 `newFruitGenerator.next()` 时， `Generator` 函数恢复执行，从第二个 `yield` 语句开始，这又使得执行暂停，同时将 `orange` 返回给调用者。
- `newFruitGenerator.next()` 的最后一次调用导致 `Generator` 函数的执行从其最后的 `yield` 恢复，一个返回语句，它终止 `Generator` 函数，返回 `watermelon` ，并将结果对象中的 `done` 属性设置为 `true` 。

Generators作为迭代器

为了更好地理解为什么 `Generator` 函数对实现迭代器非常有用，我们来构建一个例子。在我们将调用 `iteratorGenerator.js` 的新模块中，我们编写下面的代码：

```
function* iteratorGenerator(arr) {
  for (let i = 0; i < arr.length; i++) {
    yield arr[i];
  }
}
const iterator = iteratorGenerator(['apple', 'orange', 'watermel
on']);
let currentItem = iterator.next();
while (!currentItem.done) {
  console.log(currentItem.value);
  currentItem = iterator.next();
}
```

此代码应按如下所示打印数组中的元素：

```
apple
orange
watermelon
```

在这个例子中，每次我们调用 `iterator.next()` 时，我们都会恢复 `Generator` 函数的 `for` 循环，通过 `yield` 数组中的下一个项来运行另一个循环。这演示了如何在函数调用过程中维护 `Generator` 的状态。当继续执行时，循环和所有变量的值与 `Generator` 函数执行暂停时的状态完全相同。

传值给Generators

现在我们继续研究 `Generator` 的基本功能，首先学习如何将值传递回 `Generator` 函数。这其实很简单，我们需要做的只是为 `next()` 方法提供一个参数，并且该值将作为 `Generator` 函数内的 `yield` 语句的返回值提供。

为了展示这一点，我们来创建一个新的简单模块：

```
function* twoWayGenerator() {
  const what = yield null;
  console.log('Hello ' + what);
}
const twoWay = twoWayGenerator();
twoWay.next();
twoWay.next('world');
```

当执行时，前面的代码会输出 `Hello world` 。我们做如下的解释：

- 第一次调用 `next()` 方法时，`Generator` 函数到达第一个 `yield` 语句，然后暂停。
- 当 `next('world')` 被调用时，`Generator` 函数从上次停止的位置，也就是上次的 `yield` 语句点恢复，但是这次我们有一个值传递到 `Generator` 函数。这个值将被赋值到 `what` 变量。生成器然后执行 `console.log()` 指令并终止。

用类似的方式，我们可以强制 `Generator` 函数抛出异常。这可以通过使用 `Generator` 函数的 `throw` 方法来实现，如下例所示：

```
const twoWay = twoWayGenerator();
twoWay.next();
twoWay.throw(new Error());
```

在这个最后这段代码，`twoWayGenerator()` 函数将在 `yield` 函数返回的时候抛出异常。这就好像从 `Generator` 函数内部抛出了一个异常一样，这意味着它可以像使用 `try ... catch` 块一样进行捕获和处理异常。

Generator实现异步控制流

你一定想知道 `Generator` 函数如何帮助我们处理异步操作。我们可以通过创建一个接受 `Generator` 函数作为参数的特殊函数来演示这一点，并允许我们在 `Generator` 函数内部使用异步代码。这个函数在异步操作完成时要注意恢复 `Generator` 函数的执行。我们将调用这个函数 `asyncFlow()`：

```
function asyncFlow(generatorFunction) {
  function callback(err) {
    if (err) {
      return generator.throw(err);
    }
    const results = [].slice.call(arguments, 1);
    generator.next(results.length > 1 ? results : results[0]);
  }
  const generator = generatorFunction(callback);
  generator.next();
}
```

前面的函数取一个 `Generator` 函数作为输入，然后立即调用：

```
const generator = generatorFunction(callback);
generator.next();
```

`generatorFunction()` 接受一个特殊的回调函数作为参数，当 `generator.throw()` 如果接收到一个错误，便立即返回。另外，通过将在回调函数中接收的 `results` 传值回 `Generator` 函数继续 `Generator` 函数的执行:

```
if (err) {
  return generator.throw(err);
}
const results = [].slice.call(arguments, 1);
generator.next(results.length > 1 ? results : results[0]);
```

为了说明这个简单的辅助函数的强大，我们创建一个叫做 `clone.js` 的新模块，这个模块只是创建它本身的克隆。粘贴我们刚才创建的 `asyncFlow()` 函数，核心代码如下：

```
const fs = require('fs');
const path = require('path');
asyncFlow(function*(callback) {
  const fileName = path.basename(__filename);
  const myself = yield fs.readFile(fileName, 'utf8', callback);
  yield fs.writeFile(`clone_of_${filename}`, myself, callback);
  console.log('Clone created');
});
```

明显地，有了 `asyncFlow()` 函数的帮助，我们可以像我们书写同步阻塞函数一样用同步的方式来书写异步代码了。并且这个结果背后的原理显得很清楚。一旦异步操作结束，传递给每个异步函数的回调函数将继续 `Generator` 函数的执行。没有什么复杂的，但是结果确实很令人意外。

这个技术有其他两个变化，一个是 `Promise` 的使用，另外一个则是 `thunks` 。

> 在基于Generator的控制流中使用的thunk只是一个简单的函数，它除了回调之外，部分地应用了原始函数的所有参数。返回值是另一个只接受回调作为参数的函数。例如，fs.readFile（）的thunkified版本如下所示：

```
function readFileThunk(filename, options) {
  return function(callback) {
    fs.readFile(filename, options, callback);
  }
}
```

`thunk` 和 `Promise` 都允许我们创建不需要回调的 `Generator` 函数作为参数传递，例如，使用 `thunk` 的 `asyncFlow()` 版本如下：

```
function asyncFlowWithThunks(generatorFunction) {
  function callback(err) {
    if (err) {
      return generator.throw(err);
    }
    const results = [].slice.call(arguments, 1);
    const thunk = generator.next(results.length > 1 ? results :
results[0]).value;
    thunk && thunk(callback);
  }
  const generator = generatorFunction();
  const thunk = generator.next().value;
  thunk && thunk(callback);
}
```

这个技巧是读取 `generator.next()` 的返回值，返回值中包含 `thunk` 。下一步是通过注入特殊的回调函数调用 `thunk` 本身。这允许我们写下面的代码：

```
asyncFlowWithThunk(function*() {
  const fileName = path.basename(__filename);
  const myself = yield readFileThunk(__filename, 'utf8');
  yield writeFileThunk(`clone_of_${fileName}`, myself);
  console.log("Clone created")
});
```

使用co的基于Gernator的控制流

你应该已经猜到了， `Node.js` 生态系统会借助 `Generator` 函数来提供一些处理异步控制流的解决方案，例如，suspend是其中一个最老的支持 `Promise` 、 `thunks` 和 `Node.js` 风格回调函数和正常风格的回调函数的 库。还有，大部分我们之前分析的 `Promise` 库都提供工具函数使得 `Generator` 和 `Promise` 可以一起使用。

我们选择co作为本章节的例子。它支持很多类型的 `yieldables` ，其中一些是：

- `Thunks`
- `Promises`
- `Arrays` (并行执行)
- `Objects` (并行执行)
- `Generators` (委托)
- `Generator` 函数(委托)

还有很多框架或库是基于 `co` 生态系统的，包括以下一些：

- `Web框架` ，最流行的是koa
- 实现特定控制流模式的库
- 包装流行的 `API` 兼容 `co` 的库

我们使用 `co` 重新实现我们的 `Generator` 版本的 `Web爬虫应用程序` 。

为了将 `Node.js` 风格的函数转换成 `thunks` ，我们将会使用一个叫做thunkify的库。

顺序执行

让我们通过修改 `Web爬虫应用程序` 的版本2开始我们对 `Generator` 函数和 `co` 的实际探索。我们要做的第一件事就是加载我们的依赖包，并生成我们要使用的函数的 `thunkified` 版本。这些将在 `spider.js` 模块的最开始进行：

```
const thunkify = require('thunkify');
const co = require('co');
const request = thunkify(require('request'));
const fs = require('fs');
const mkdirp = thunkify(require('mkdirp'));
const readFile = thunkify(fs.readFile);
const writeFile = thunkify(fs.writeFile);
const nextTick = thunkify(process.nextTick);
```

看上述代码，我们可以注意到与本章前面 `promisify` 化的 `API` 的代码的一些相似之处。在这一点上，有意思的是，如果我们使用我们的 `promisified` 版本的函数来代替 `thunkified` 的版本，代码将保持完全一样，这要归功于 `co` 支持 `thunk` 和 `Promise` 对象作为 `yieldable` 对象。事实上，如果我们想，甚至可以在同一个应用程序中使用 `thunk` 和 `Promise` ，即使在同一个 `Generator` 函数中。就灵活性而言，这是一个巨大的优势，因为它使我们能够使用基于 `Generator` 函数的控制流来解决我们应用程序中的问题。

好的，现在让我们开始将 `download()` 函数转换为一个 `Generator` 函数：

```
function* download(url, filename) {
  console.log(`Downloading ${url}`);
  const response = yield request(url);
  const body = response[1];
  yield mkdirp(path.dirname(filename));
  yield writeFile(filename, body);
  console.log(`Downloaded and saved ${url}`);
  return body;
}
```

通过使用 `Generator` 和 `co` ，我们的 `download()` 函数变得简单多了。当我们需要做异步操作的时候，我们使用异步的 `Generator` 函数作为 `thunk` 来把之前的内容转化到 `Generator` 函数，并使用 `yield` 子句。

然后我们开始实现我们的 `spider()` 函数：

```
function* spider(url, nesting) {
  cost filename = utilities.urlToFilename(url);
  let body;
  try {
    body = yield readFile(filename, 'utf8');
  } catch (err) {
    if (err.code !== 'ENOENT') {
      throw err;
    }
    body = yield download(url, filename);
  }
  yield spiderLinks(url, body, nesting);
}
```

从上述代码中一个有趣的细节是我们可以使用 `try...catch` 语句块来处理异常。我们还可以使用 `throw` 来传播异常。另外一个细节是我们 `yield` 我们的 `download()` 函数，而这个函数既不是一个 `thunk` ，也不是一个 `promisified` 函数，只是另外的一个 `Generator` 函数。这也毫无问题，由于 `co` 也支持其他 `Generators` 作为 `yieldables` 。

最后转换 `spiderLinks()` ，在这个函数中，我们递归下载一个网页的链接。在这个函数中使用 `Generators` ，显得简单多了：

```
function* spiderLinks(currentUrl, body, nesting) {
  if (nesting === 0) {
    return nextTick();
  }
  const links = utilities.getPageLinks(currentUrl, body);
  for (let i = 0; i < links.length; i++) {
    yield spider(links[i], nesting - 1);
  }
}
```

看上述代码。虽然顺序迭代没有什么模式可以展示。 `Generator` 和 `co` 辅助我们做了很多，方便了我们可以使用同步方式开书写异步代码。

看最重要的部分，程序的入口：

```
co(function*() {
  try {
    yield spider(process.argv[2], 1);
    console.log(`Download complete`);
  } catch (err) {
    console.log(err);
  }
});
```

这是唯一一处需要调用 `co(...)` 来封装的一个 `Generator` 。实际上，一旦我们这么做， `co` 会自动封装我们传递给 `yield` 语句的任何 `Generator` 函数，并且这个过程是递归的，所以程序的剩余部分与我们是否使用 `co` 是完全无关的，虽然是被 `co` 封装在里面。

现在应该可以运行使用 `Generator` 函数改写的 `Web爬虫应用程序` 了。

并行执行

不幸的是，虽然 `Generator` 很方便地进行顺序执行，但是不能直接用来并行化执行一组任务，至少不能仅仅使用 `yield` 和 `Generator` 。之前，在种情况下我们使用的模式只是简单地依赖于一个基于回调或者 `Promise` 的函数，但使用了 `Generator` 函数后，一切会显得更简单。

幸运的是，如果不限制并发数的并行执行， `co` 已经可以通过 `yield` 一个 `Promise` 对象、 `thunk` 、 `Generator` 函数，甚至包含 `Generator` 函数的数组来实现。

考虑到这一点，我们的 `Web爬虫应用程序` 第三版可以通过重写 `spiderLinks()` 函数来做如下改动：

```
function* spiderLinks(currentUrl, body, nesting) {
  if (nesting === 0) {
    return nextTick();
  }
  const links = utilities.getPageLinks(currentUrl, body);
  const tasks = links.map(link => spider(link, nesting - 1));
  yield tasks;
}
```

但是上述函数所做的只是拿到所有的任务，这些任务本质上都是通过 `Generator` 函数来实现异步的，如果在 `co` 的 `thunk` 内对一个包含 `Generator` 函数的数组使用 `yield` ，这些任务都会并行执行。外层的 `Generator` 函数会等到 `yield` 子句的所有异步任务并行执行后再继续执行。

接下来我们看怎么用一个基于回调函数的方式来解决相同的并行流。我们用这种方式重写 `spiderLinks()` 函数：

```
function spiderLinks(currentUrl, body, nesting) {
  if (nesting === 0) {
    return nextTick();
  }
  // 返回一个thunk
  return callback => {
    let completed = 0,
      hasErrors = false;
    const links = utilities.getPageLinks(currentUrl, body);
    if (links.length === 0) {
      return process.nextTick(callback);
    }

    function done(err, result) {
      if (err && !hasErrors) {
        hasErrors = true;
        return callback(err);
      }
      if (++completed === links.length && !hasErrors) {
        callback();
      }
    }
    for (let i = 0; i < links.length; i++) {
      co(spider(links[i], nesting - 1)).then(done);
    }
  }
}
```

我们使用 `co` 并行运行 `spider()` 函数，调用 `Generator` 函数返回了一个 `Promise` 对象。这样，等待 `Promise` 完成后调用 `done()` 函数。通常，基于 `Generator` 控制流的库都有这一功能，因此如果需要，你总是可以将一个 `Generator` 转换成一个基于回调或基于 `Promise` 的函数。

为了并行开启多个下载任务，我们只要重用在前面定义的基于回调的并行执行的模式。我们应该也注意到我们将 `spiderLinks()` 转换成一个 `thunk` (而不再是一个 `Generator` 函数)。这使得当全部并行任务完成时，我们有一个回调函数可以调用。

> 上面讲到的是将一个Generator函数转换为一个thunk的模式，使之能够支持其他的基于回调或基于Promise的控制流算法，并可以通过同步阻塞的代码风格书写异步代码。

限制并行执行

现在我们知道如何处理异步执行流程，应该很容易规划我们的 `Web爬虫应用程序` 的第四版的实现，这个版本对并发下载任务的数量施加了限制。我们有几个方案可以用来做到这一点。其中一些方案如下：

- 使用先前实现的基于回调的 `TaskQueue` 类。我们只需要 `thunkify` 我们

的 `Generator` 函数和其提供的回调函数即可。
- 使用基于 `Promise` 的 `TaskQueue` 类，并确保每个作为任务的 `Generator` 函数都被转换成一个返回 `Promise` 对象的函数。
- 使用 `async` ， `thunkify` 我们打算使用的工具函数，此外还需要把我们用到的 `Generator` 函数转化为基于回调的模式，以便于能够被这个库较好地使用。
- 使用基于 `co` 的生态系统中的库，特别是专门为这种场景的库，如co-limiter。
- 实现基于生产者 - 消费者模型的自定义算法，这与 `co-limiter` 的内部实现原理相同。

为了学习，我们选择最后一个方案，甚至帮助我们可以更好地理解一种经常与协程(也和线程和进程)同步相关的模式。

生产者 - 消费者模式

我们的目标是利用队列来提供固定数量的 `workers` ，与我们想要设置的并发级别一样多。为了实现这个算法，我们将基于本章前面定义的 `TaskQueue` 类改写：

```
class TaskQueue {
  constructor(concurrency) {
    this.concurrency = concurrency;
    this.running = 0;
    this.taskQueue = [];
    this.consumerQueue = [];
    this.spawnWorkers(concurrency);
  }
  pushTask(task) {
    if (this.consumerQueue.length !== 0) {
      this.consumerQueue.shift()(null, task);
    } else {
      this.taskQueue.push(task);
    }
  }
  spawnWorkers(concurrency) {
    const self = this;
    for (let i = 0; i < concurrency; i++) {
      co(function*() {
        while (true) {
          const task = yield self.nextTask();
          yield task;
        }
      });
    }
  }
  nextTask() {
    return callback => {
      if (this.taskQueue.length !== 0) {
        return callback(null, this.taskQueue.shift());
      }
      this.consumerQueue.push(callback);
    }
  }
}
```

让我们分析这个 `TaskQueue` 类的新实现。首先是在构造函数中。需要调用一次 `this.spawnWorkers()` ，因为这是启动 `worker` 的方法。

我们的 `worker` 很简单，它们只是用 `co()` 包装的立即执行的 `Generator` 函数，所以每个 `Generator` 函数可以并行执行。在内部，每个 `worker` 正在运行在一个死循环（ `while(true){}` ）中，一直阻塞（ `yield` ）到新任务在队列中可用时（ `yield self.nextTask()` ），一旦可以执行新任务， `yield` 这个异步任务直到其完成。您可能想知道我们如何能够限制并行执行，并让下一个任务在队列中处于等待状态。答案是在 `nextTask()` 方法中。我们来详细地看看在这个方法的原理：

```
nextTask() {
  return callback => {
    if (this.taskQueue.length !== 0) {
      return callback(null, this.taskQueue.shift());
    }
    this.consumerQueue.push(callback);
  }
}
```

我们看这个函数内部发生了什么，这才是这个模式的核心：

1. 这个方法返回一个对于 `co` 而言是一个合法的 `yieldable` 的 `thunk` 。
2. 只要 `taskQueue` 类生成的实例中还有下一个任务， `thunk` 的回调函数会被立即调用。回调函数调用时，立马解锁一个 `worker` 的阻塞状态， `yield` 这一个任务。
3. 如果队列中没有任务了，回调函数本身会被放入 `consumerQueue` 中。通过这种做法，我们将一个 `worker` 置于空闲（ `idle` ）的模式。一旦我们有一个新的任务来要处理，在 `consumerQueue` 队列中的回调函数会被调用，立马唤醒我们这一 `worker` 进行异步处理。

现在，为了理解 `consumerQueue` 队列中的空闲 `worker` 是如何恢复工作的，我们需要分析 `pushTask()` 方法。如果当前有回调函数可用的话， `pushTask()` 方法将调用 `consumerQueue` 队列中的第一个回调函数，从而将取消对 `worker` 的锁定。如果没有可用的回调函数，这意味着所有的 `worker` 都是工作状态，只需要添加一个新的任务到 `taskQueue` 任务队列中。

在 `TaskQueue` 类中， `worker` 充当消费者的角色，而调用 `pushTask()` 函数的角色可以被认为是生产者。这个模式向我们展示了一个 `Generator` 函数实际上可以跟一个线程或进程类似。实际上，生产者-消费者之间问题是研究进程间通信和同步时最常见的问题，但正如我们已经提到的那样，它对于进程和线程来说，也是一个常见的例子。

限制下载任务的并发量

既然我们已经使用 `Generator` 函数和生产者-消费者模型实现一个限制并行算法，并且已经在 `Web爬虫应用程序` 第四版应用它来限制中下载任务的并发数。 首先，我们加载和初始化一个 `TaskQueue` 对象：

```
const TaskQueue = require('./taskQueue');
const downloadQueue = new TaskQueue(2);
```

然后，修改 `spiderLinks()` 函数。和之前不限制并发的版本类似，所以这里我们只展示修改的部分，主要是通过调用新版本的 `TaskQueue` 类生成的实例的 `pushTask()` 方法来限制并行执行：

```
function spiderLinks(currentUrl, body, nesting) {
  //...
  return (callback) => {
    //...
    function done(err, result) {
      //...
    }
    links.forEach(function(link) {
      downloadQueue.pushTask(function*() {
        yield spider(link, nesting - 1);
        done();
      });
    });
  }
}
```

在每个任务中，我们在下载完成后立即调用 `done()` 函数，因此我们可以计算下载了多少个链接，然后在完成下载时通知 `thunk` 的回调函数执行。

配合Babel使用Async await新语法

回调函数、 `Promise` 和 `Generator` 函数都是用于处理 `JavaScript` 和 `Node.js` 异步问题的方式。正如我们所看到的， `Generator` 的真正意义在于它提供了一种方式来暂停一个函数的执行，然后等待前面的任务完成后再继续执行。我们可以使用这样的特性来书写异步代码，并且让开发者用同步阻塞的代码风格来书写异步代码。等到异步操作的结果返回后才恢复当前函数的执行。

但 `Generator` 函数是更多的是用来处理迭代器，然而迭代器在异步代码的使用显得有点笨重。代码可能难以理解，导致代码易读性和可维护性差。

但在不远的将来会有一种更加简洁的语法。实际上，这个提议即将引入到 `ESMASCript 2017` 的规范中，这项规范定义了 `async` 函数语法。

`async` 函数规范引入两个关键字(`async` 和 `await`)到原生的 `JavaScript` 语言中，改进我们书写异步代码的方式。

为了理解这项语法的用法和优势为，我们看一个简单的例子：

```
const request = require('request');

function getPageHtml(url) {
  return new Promise(function(resolve, reject) {
    request(url, function(error, response, body) {
      resolve(body);
    });
  });
}
async function main() {
  const html = await getPageHtml('http://google.com');
  console.log(html);
}

main();
console.log('Loading...');
```

在上述代码中，有两个函数： `getPageHtml` 和 `main` 。第一个函数的作用是提取给定 `URL` 的一个远程网页的 `HTML` 文档代码。值得注意的是，这个函数返回一个 `Promise` 对象。

重点在于 `main` 函数，因为在这里使用了 `async` 和 `await` 关键字。首先要注意的是函数要以 `async` 关键字为前缀。意思是这个函数执行的是异步代码并且允许它在函数体内使用 `await` 关键字。 `await` 关键字在 `getPageHtml` 调用之前，告诉 `JavaScript` 解释器在继续执行下一条指令之前，等待 `getPageHtml` 返回的 `Promise` 对象的结果。这样， `main` 函数内部哪部分代码是异步的，它会等待异步代码的完成再继续执行后续操作，并且不会阻塞这段程序其余部分的正常执行。实际上，控制台会打印字符串 `Loading...` ，随后是Google主页的 `HTML` 代码。

是不是这种方法的可读性更好并且更容易理解呢? 不幸地是，这个提议尚未定案，即使通过这个提议，我们需要等下一个版本 的 `ECMAScript` 规范出来并把它集成到 `Node.js` 后，才能使用这个新语法。 所以我们今天做了什么?只是漫无目的地等待?不是，当然不是！我们已经可以在我们的代码中使用 `async await` 语法，只要我们使用 `Babel` 。

安装与运行Babel

`Babel` 是一个 `JavaScript` 编译器(或翻译器)，能够使用语法转换器将高版本的 `JavaScript` 代码转换成其他 `JavaScript` 代码。语法转换器允许例如我们书写并使用 `ES2015` ， `ES2016` ， `JSX` 和其它的新语法，来翻译成往后兼容的代码，在 `JavaScript` 运行环境如浏览器或 `Node.js` 中都可以使用 `Babel` 。

在项目中使用 `npm` 安装 `Babel` ，命令如下：

```
npm install --save-dev babel-cli
```

我们还需要安装插件以支持`async await`语法的解释或翻译：

```
npm install --save-dev babel-plugin-syntax-async-functions babel
-plugin-tranform-async-to-generator
```

现在假设我们想运行我们之前的例子（称为`index.js`）。我们需要通过以下命令启动：

```
node_modules/.bin/babel-node --plugins "syntax-async-functions,t
ransform-async-to-generator" index.js
```

这样，我们使用支持`async await`的转换器动态地转换源代码。`Node.js`运行的实际是保存在内存中的往后兼容的代码。

`Babel`也能被配置为一个代码构建工具，保存翻译或解释后的代码到本地文件系统中，便于我们部署和运行生成的代码。

> 关于如何安装和配置Babel，可以到官方网站 https://babeljs.io 查阅相关文档。

几种方式的比较

现在，我们应该对于怎么处理`JavaScript`的异步问题有了一个更好的认识和总结。在下面的表格中总结几大机制的优势和劣势：

方案	Pros	Cons
扁平的 JavaScript（Plain JavaScript）	● 不需要任何库或技术 ● 提供最好的性能 ● 提供与第三方库最佳的兼容性 ● 允许 ad hoc 和更高级算法的创建	可能需要额外的代码和相对复杂的算法
Async（library）	● 简化最常见的控制流模式 ● 还是一个 callback-based 的解决方案 ● 性能好	● 引入一个外部依赖 ● 对于高级的流来说还是不够的
Promises	● 大大简化最常见的控制流模式 ● 鲁棒的 error 处理 ● ES2015 规范的一部分 ● 确保 onFullfilled 和 onRejected 的延迟调用	● 需要 promisify callback-based 的 APIs ● 引入以小的性能损失
Generators	● 使得非阻塞 API 看起来像阻塞一样 ● ES2015 规范的一部分	● 需要一个辅助的控制流库 ● 依然需要 callbacks 或 promises 来实现非顺序流 ● 需要 thunkify 或 promisify 非 generator-based 的 APIs
Async await	● 使得非阻塞 API 看起来像阻塞一样 ● 简洁直观的语法	● 在原生的 JavaScript 和 Node.js 还不能使用 ● 今天使用需要 Babel 或其他翻译器和一些配置

> 值得一提的是，我们选择在本章中仅介绍处理异步控制流程的最受欢迎的解决方案，或者是广泛使用的解决方案，但是例如Fibers（https://npmjs.org/package/fibers）和Streamline（https://npmjs.org/package/streamline）也是值得一看的。

总结

在本章中，我们分析了一些处理异步控制流的方法，分析了 `Promise` 、 `Generator` 函数和即将到来的 `async await` 语法。

我们学习了如何使用这些方法编写更简洁，更具有可读性的异步代码。我们讨论了这些方法的一些最重要的优点和缺点，并认识到即使它们非常有用，也需要一些时间来掌握。这就是这几种方式也没有完全取代在许多情况下仍然非常有用的回调的原因。作为一名开发人员，应该按照实际情况分析决定使用哪种解决方案。如果您正在构建执行异步操作的公共库，则应该提供易于使用的 `API` ，即使对于只想使用回调的开发人员也是如此。

在下一章中，我们将探讨另一个与异步代码执行相关的机制，这也是整个 `Node.js` 生态系统中的另一个基本构建块： `streams` 。

Coding with Streams

`Streams` 是 `Node.js` 最重要的组件和模式之一。 社区中有一句格言“Stream all the things（Steam就是所有的）”，仅此一点就足以描述流在 `Node.js` 中的地位。 `Dominic Tarr` 作为 `Node.js` 社区的最大贡献者，它将流定义为 `Node.js` 最好，也是最难以理解的概念。

使 `Node.js` 的 `Streams` 如此吸引人还有其它原因; 此外， `Streams` 不仅与性能或效率等技术特性有关，更重要的是它们的优雅性以及它们与 `Node.js` 的设计理念完美契合的方式。

在本章中，将会学到以下内容：

- `Streams` 对于 `Node.js` 的重要性。
- 如何创建并使用 `Streams` 。
- `Streams` 作为编程范式，不只是对于 `I/O` 而言，在多种应用场景下它的应用和强大的功能。
- 管道模式和在不同的配置中连接 `Streams` 。

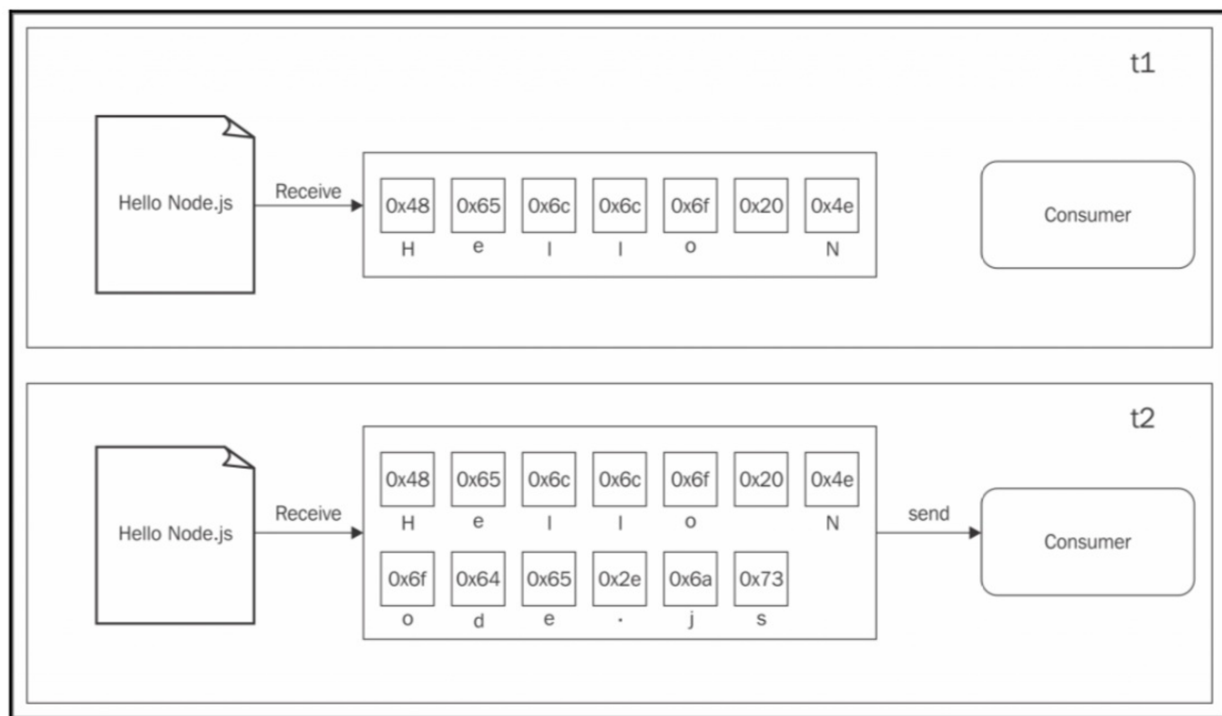
发现Streams的重要性

在基于事件的平台（如 `Node.js` ）中，处理 `I / O` 的最有效的方法是实时处理，一旦有输入的信息，立马进行处理，一旦有需要输出的结果，也立马输出反馈。

在本节中，我们将首先介绍 `Node.js` 的 `Streams` 和它的优点。 请记住，这只是一个概述，因为本章后面将会详细介绍如何使用和组合 `Streams` 。

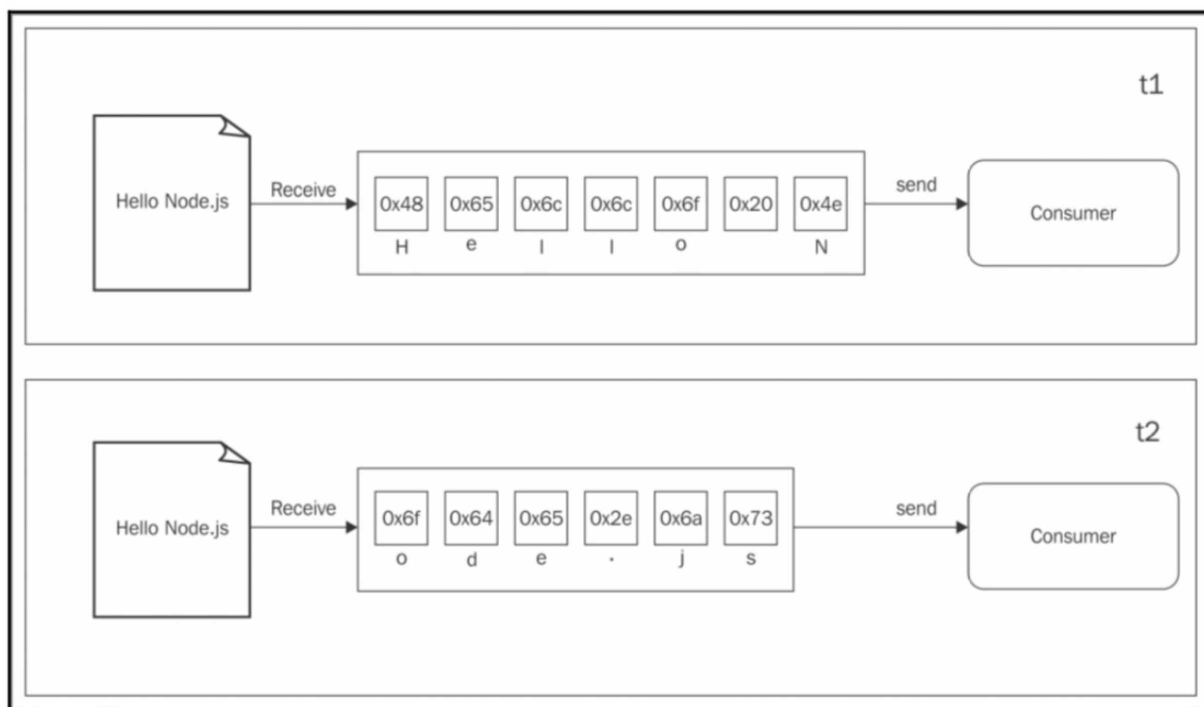
Streams和Buffer的比较

我们在本书中几乎所有看到过的异步API都是使用的 `Buffer` 模式。 对于输入操作， `Buffer` 模式会将来自资源的所有数据收集到 `Buffer` 区中; 一旦读取完整个资源，就会把结果传递给回调函数。 下图显示了这个范例的一个真实的例子：



从上图我们可以看到，在**t1**时刻，一些数据从资源接收并保存到缓冲区。 在**t2**时刻，最后一段数据被接收到另一个数据块，完成读取操作，这时，把整个缓冲区的内容发送给消费者。

另一方面，`Streams`允许你在数据到达时立即处理数据。 如下图所示：



这一张图显示了`Streams`如何从资源接收每个新的数据块，并立即提供给消费者，消费者现在不必等待缓冲区中收集所有数据再处理每个数据块。

但是这两种方法有什么区别呢？ 我们可以将它们概括为两点：

- 空间效率
- 时间效率

此外， `Node.js` 的 `Streams` 具有另一个重要的优点：可组合性（**composability**）。现在让我们看看这些属性对我们设计和编写应用程序的方式会产生什么影响。

空间效率

首先， `Streams` 允许我们做一些看起来不可能的事情，通过缓冲数据并一次性处理。例如，考虑一下我们必须读取一个非常大的文件，比如说数百 `MB` 甚至千 `MB` 。显然，等待完全读取文件时返回大 `Buffer` 的 `API` 不是一个好主意。想象一下，如果并发读取一些大文件，我们的应用程序很容易耗尽内存。除此之外， `V8` 中的 `Buffer` 不能大于 `0x3FFFFFFF` 字节（小于 `1GB` ）。所以，在耗尽物理内存之前，我们可能会碰壁。

使用**Buffered**的**API**进行压缩文件

举一个具体的例子，让我们考虑一个简单的命令行接口（ `CLI` ）的应用程序，它使用 `Gzip` 格式压缩文件。使用 `Buffered` 的 `API` ，这样的应用程序在 `Node.js` 中大概这么编写（为简洁起见，省略了异常处理）：

```
const fs = require('fs');
const zlib = require('zlib');
const file = process.argv[2];
fs.readFile(file, (err, buffer) => {
  zlib.gzip(buffer, (err, buffer) => {
    fs.writeFile(file + '.gz', buffer, err => {
      console.log('File successfully compressed');
    });
  });
});
```

现在，我们可以尝试将前面的代码放在一个叫做 `gzip.js` 的文件中，然后执行下面的命令：

```
node gzip <path to file>
```

如果我们选择一个足够大的文件，比如说大于 `1GB` 的文件，我们会收到一个错误信息，说明我们要读取的文件大于最大允许的缓冲区大小，如下所示：

```
RangeError: File size is greater than possible Buffer:0x3FFFFFFF
```

```
➜ Node.js-Code node test.js test.js
File successfully compressed
➜ Node.js-Code node test.js ~/Downloads/Microsoft_Office_2016_15.40.17110800_Installer.pkg
File successfully compressed
```

> 上面的例子中，没找到一个大文件，但确实对于大文件的读取速率慢了许多。

正如我们所预料到的那样，使用 `Buffer` 来进行大文件的读取显然是错误的。

使用**Streams**进行压缩文件

我们必须修复我们的 `Gzip` 应用程序，并使其处理大文件的最简单方法是使用 `Streams` 的 `API` 。让我们看看如何实现这一点。 让我们用下面的代码替换刚创建的模块的内容：

```
const fs = require('fs');
const zlib = require('zlib');
const file = process.argv[2];
fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream(file + '.gz'))
  .on('finish', () => console.log('File successfully compressed'
));
```

“是吗？”你可能会问。是的；正如我们所说的，由于 `Streams` 的接口和可组合性，因此我们还能写出这样的更加简洁，优雅和精炼的代码。 我们稍后会详细地看到这一点，但是现在需要认识到的重要一点是，程序可以顺畅地运行在任何大小的文件上，理想情况是内存利用率不变。 尝试一下（但考虑压缩一个大文件可能需要一段时间）。

时间效率

现在让我们考虑一个压缩文件并将其上传到远程 `HTTP` 服务器的应用程序的例子，该远程 `HTTP` 服务器进而将其解压缩并保存到文件系统中。如果我们的客户端是使用 `Buffered` 的 `API` 实现的，那么只有当整个文件被读取和压缩时，上传才会开始。 另一方面，只有在接收到所有数据的情况下，解压缩才会在服务器上启动。 实现相同结果的更好的解决方案涉及使用 `Streams` 。 在客户端机器上， `Streams` 只要从文件系统中读取就可以压缩和发送数据块，而在服务器上，只要从远程对端接收到数据块，就可以解压每个数据块。 我们通过构建前面提到的应用程序来展示这一点，从服务器端开始。

我们创建一个叫做 `gzipReceive.js` 的模块，代码如下：

```
const http = require('http');
const fs = require('fs');
const zlib = require('zlib');

const server = http.createServer((req, res) => {
  const filename = req.headers.filename;
  console.log('File request received: ' + filename);
  req
    .pipe(zlib.createGunzip())
    .pipe(fs.createWriteStream(filename))
    .on('finish', () => {
      res.writeHead(201, {
        'Content-Type': 'text/plain'
      });
      res.end('That\'s it\n');
      console.log(`File saved: ${filename}`);
    });
});

server.listen(3000, () => console.log('Listening'));
```

服务器从网络接收数据块，将其解压缩，并在接收到数据块后立即保存，这要归功于 `Node.js` 的 `Streams` 。

我们的应用程序的客户端将进入一个名为 `gzipSend.js` 的模块，如下所示：

在前面的代码中，我们再次使用 `Streams` 从文件中读取数据，然后在从文件系统中读取的同时压缩并发送每个数据块。

现在，运行这个应用程序，我们首先使用以下命令启动服务器：

```
node gzipReceive
```

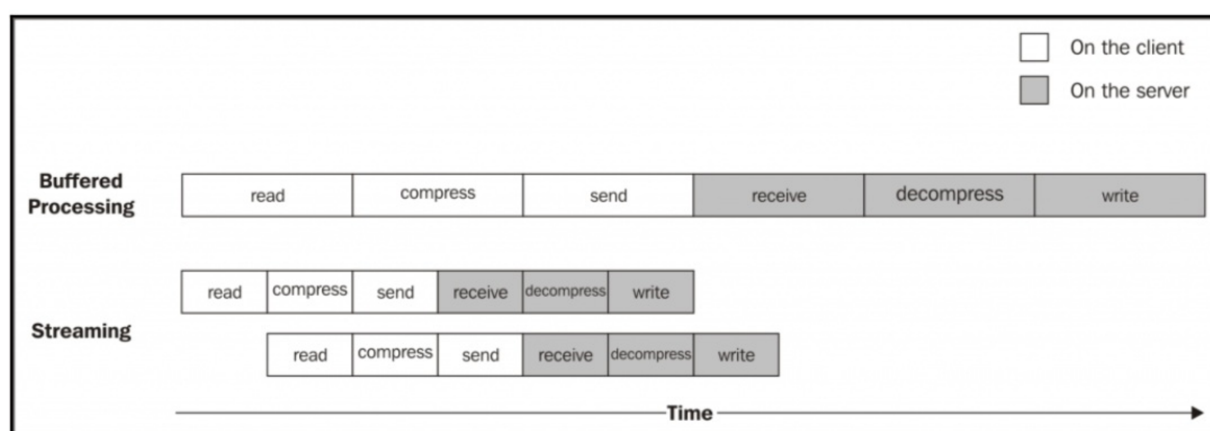
然后，我们可以通过指定要发送的文件和服务器的地址（例如 `localhost` ）来启动客户端：

```
node gzipSend <path to file> localhost
```

```
../Node.js-Code (zsh)
➜ Node.js-Code node gzipSend.js gzipSend.js localhost
File successfully sent
Server response: 201
➜ Node.js-Code ls
gzipReceive.js gzipSend.js
➜ Node.js-Code
```

```
node (node)
➜ Node.js-Code ls
gzipReceive.js gzipSend.js
➜ Node.js-Code node gzipReceive.js
Listening
File request received: gzipSend.js
File saved: gzipSend.js
```

如果我们选择一个足够大的文件，我们将更容易地看到数据如何从客户端流向服务器，但为什么这种模式下，我们使用 `Streams` ，比使用 `Buffered` 的 `API` 更有效率？ 下图应该给我们一个提示：



一个文件被处理的过程，它经过以下阶段：

1. 客户端从文件系统中读取
2. 客户端压缩数据
3. 客户端将数据发送到服务器
4. 服务端接收数据
5. 服务端解压数据
6. 服务端将数据写入磁盘

为了完成处理，我们必须按照流水线顺序那样经过每个阶段，直到最后。在上图中，我们可以看到，使用 `Buffered` 的 `API` ，这个过程完全是顺序的。为了压缩数据，我们首先必须等待整个文件被读取完毕，然后，发送数据，我们必须等待整个文件被读取和压缩，依此类推。当我们使用 `Streams` 时，只要我们收到第一个数据块，流水线就会被启动，而不需要等待整个文件的读取。但更令人惊讶的是，当下一块数据可用时，不需要等待上一组任务完成；相反，另一条装配线是并行启动的。因为我们执行的每个任务都是异步的，这样显得很完美，所以可以通过 `Node.js` 来并行执行 `Streams` 的相关操作；唯一的限制就是每个阶段都必须保证数据块的到达顺序。

从前面的图可以看出，使用 `Streams` 的结果是整个过程花费的时间更少，因为我们不用等待所有数据被全部读取完毕和处理。

组合性

到目前为止，我们已经看到的代码已经告诉我们如何使用 `pipe()` 方法来组装 `Streams` 的数据块， `Streams` 允许我们连接不同的处理单元，每个处理单元负责单一的职责（这是符合 `Node.js` 风格的）。这是可能的，因为 `Streams` 具有统一的接口，并且就 `API` 而言，不同 `Streams` 也可以很好的进行交互。唯一的先决条件是管道的下一个 `Streams` 必须支持上一个 `Streams` 生成的数据类型，可以是二进制，文本甚至是对象，我们将在后面的章节中看到。

为了证明 `Streams` 组合性的优势，我们可以尝试在我们先前构建的 `gzipReceive / gzipSend` 应用程序中添加加密功能。 为此，我们只需要通过向流水线添加另一个 `Streams` 来更新客户端。 确切地说，由 `crypto.createChipher()` 返回的流。 由此产生的代码应如下所示：

```
const fs = require('fs');
const zlib = require('zlib');
const crypto = require('crypto');
const http = require('http');
const path = require('path');

const file = process.argv[2];
const server = process.argv[3];

const options = {
  hostname: server,
  port: 3000,
  path: '/',
  method: 'PUT',
  headers: {
    filename: path.basename(file),
    'Content-Type': 'application/octet-stream',
    'Content-Encoding': 'gzip'
  }
};

const req = http.request(options, res => {
  console.log('Server response: ' + res.statusCode);
});

fs.createReadStream(file)
  .pipe(zlib.createGzip())
  .pipe(crypto.createCipher('aes192', 'a_shared_secret'))
  .pipe(req)
  .on('finish', () => {
    console.log('File successfully sent');
  });
```

使用相同的方式，我们更新服务端的代码，使得它可以在数据块进行解压之前先解密：

```
const http = require('http');
const fs = require('fs');
const zlib = require('zlib');
const crypto = require('crypto');

const server = http.createServer((req, res) => {
  const filename = req.headers.filename;
  console.log('File request received: ' + filename);
  req
    .pipe(crypto.createDecipher('aes192', 'a_shared_secret'))
    .pipe(zlib.createGunzip())
    .pipe(fs.createWriteStream(filename))
    .on('finish', () => {
      res.writeHead(201, {
        'Content-Type': 'text/plain'
      });
      res.end('That\'s it\n');
      console.log(`File saved: ${filename}`);
    });
});

server.listen(3000, () => console.log('Listening'));
```

> crypto是Node.js的核心模块之一，提供了一系列加密算法。

只需几行代码，我们就在应用程序中添加了一个加密层。 我们只需要简单地通过把已经存在的 `Streams` 模块和加密层组合到一起，就可以。类似的，我们可以添加和合并其他 `Streams` ，如同在玩乐高积木一样。

显然，这种方法的主要优点是可重用性，但正如我们从目前为止所介绍的代码中可以看到的那样， `Streams` 也可以实现更清晰，更模块化，更加简洁的代码。 出于这些原因，流通常不仅仅用于处理纯粹的 `I / O` ，而且它还是简化和模块化代码的手段。

开始使用Streams

在前面的章节中，我们了解了为什么 `Streams` 如此强大，而且它在 `Node.js` 中无处不在，甚至在 `Node.js` 的核心模块中也有其身影。 例如，我们已经看到， `fs` 模块具有用于从文件读取的 `createReadStream()` 和用于写入文件的 `createWriteStream()` ， `HTTP` 请求和响应对象本质上是 `Streams` ，并且 `zlib` 模块允许我们使用 `Streams` 式 `API` 压缩和解压缩数据块。

现在我们知道为什么 `Streams` 是如此重要，让我们退后一步，开始更详细地探索它。

Streams的结构

`Node.js` 中的每个 `Streams` 都是 `Streams` 核心模块中可用的四个基本抽象类之一的实现：

- `stream.Readable`
- `stream.Writable`
- `stream.Duplex`
- `stream.Transform`

每个 `stream` 类也是 `EventEmitter` 的一个实例。实际上， `Streams` 可以产生几种类型的事件，比如 `end` 事件会在一个可读的 `Streams` 完成读取，或者错误读取，或其过程中产生异常时触发。

> 请注意，为简洁起见，在本章介绍的例子中，我们经常会忽略适当的错误处理。但是，在生产环境下中，总是建议为所有Stream注册错误事件侦听器。

`Streams` 之所以如此灵活的原因之一是它不仅能够处理二进制数据，而且几乎可以处理任何 `JavaScript` 值。实际上， `Streams` 可以支持两种操作模式：

- 二进制模式：以数据块形式（例如 `buffers` 或 `strings` ）流式传输数据
- 对象模式：将流数据视为一系列离散对象（这使得我们几乎可以使用任何 `JavaScript` 值）

这两种操作模式使我们不仅可以使用 `I / O` 流，而且还可以作为一种工具，以函数式的风格优雅地组合处理单元，我们将在本章后面看到。

> 在本章中，我们将主要使用在Node.js 0.11中引入的Node.js流接口，也称为版本3。 有关与旧接口差异的更多详细信息，请参阅StrongLoop在https://strongloop.com/strongblog/whats-new-io-js-beta-streams3/中的优秀博客文章。

可读的Streams

一个可读的 `Streams` 表示一个数据源，在 `Node.js` 中，它使用 `stream` 模块中的 `Readableabstract` 类实现。

从Streams中读取信息

从可读 `Streams` 接收数据有两种方式： `non-flowing` 模式和 `flowing` 模式。我们来更详细地分析这些模式。

non-flowing模式（不流动模式）

从可读的 `Streams` 中读取数据的默认模式是为其附加一个可读事件侦听器，用于指示要读取的新数据的可用性。然后，在一个循环中，我们读取所有的数据，直到内部 `buffer` 被清空。这可以使用 `read()` 方法完成，该方法同步从内部缓冲区中读取数据，并返回表示数据块的 `Buffer` 或 `String` 对象。 `read()` 方法以如下使用模式：

```
readable.read([size]);
```

使用这种方法，数据随时可以直接从 `Streams` 中按需提取。

为了说明这是如何工作的，我们创建一个名为 `readStdin.js` 的新模块，它实现了一个简单的程序，它从标准输入（一个可读流）中读取数据，并将所有数据回送到标准输出：

```
process.stdin
  .on('readable', () => {
    let chunk;
    console.log('New data available');
    while ((chunk = process.stdin.read()) !== null) {
      console.log(
        `Chunk read: (${chunk.length}) "${chunk.toString()}"`
      );
    }
  })
  .on('end', () => process.stdout.write('End of stream'));
```

`read()` 方法是一个同步操作，它从可读 `Streams` 的内部 `Buffers` 区中提取数据块。如果 `Streams` 在二进制模式下工作，返回的数据块默认为一个 `Buffer` 对象。

> 在以二进制模式工作的可读的Stream中，我们可以通过在Stream上调用setEncoding(encoding)来读取字符串而不是Buffer对象，并提供有效的编码格式（例如utf8）。

数据是从可读的侦听器中读取的，只要有新的数据，就会调用这个侦听器。当内部缓冲区中没有更多数据可用时， `read()` 方法返回 `null` ；在这种情况下，我们不得不等待另一个可读的事件被触发，告诉我们可以再次读取或者等待表示 `Streams` 读取过程结束的 `end` 事件触发。当一个流以二进制模式工作时，我们也可以通过向 `read()` 方法传递一个 `size` 参数来指定我们想要读取的数据大小。这在实现网络协议或解析特定数据格式时特别有用。

现在，我们准备运行 `readStdin` 模块并进行实验。让我们在控制台中键入一些字符，然后按 `Enter` 键查看回显到标准输出中的数据。要终止流并因此生成一个正常的结束事件，我们需要插入一个 `EOF` （文件结束）字符（在 `Windows` 上使用 `Ctrl + Z` 或在 `Linux` 上使用 `Ctrl + D` ）。

我们也可以尝试将我们的程序与其他程序连接起来;这可以使用管道运算符（ `|` ），它将程序的标准输出重定向到另一个程序的标准输入。例如，我们可以运行如下命令：

```
cat <path to a file> | node readStdin
```

这是流式范例是一个通用接口的一个很好的例子，它使得我们的程序能够进行通信，而不管它们是用什么语言写的。

flowing模式（流动模式）

从 `Streams` 中读取的另一种方法是将侦听器附加到 `data` 事件；这会将 `Streams` 切换为 `flowing` 模式，其中数据不是使用 `read()` 函数来提取的，而是一旦有数据到达 `data` 监听器就被推送到监听器内。例如，我们之前创建的 `readStdin` 应用程序将使用流动模式：

```
process.stdin
  .on('data', chunk => {
    console.log('New data available');
    console.log(
      `Chunk read: (${chunk.length}) "${chunk.toString()}"`
    );
  })
  .on('end', () => process.stdout.write('End of stream'));
```

`flowing` 模式是旧版 `Streams` 接口（也称为 `Streams1` ）的继承，其灵活性较低， `API` 较少。随着 `Streams2` 接口的引入， `flowing` 模式不是默认的工作模式，要启用它，需要将侦听器附加到 `data` 事件或显式调用 `resume()` 方法。 要暂时中断 `Streams` 触发 `data` 事件，我们可以调用 `pause()` 方法，导致任何传入数据缓存在内部 `buffer` 中。

> 调用pause()不会导致Streams切换回non-flowing模式。

实现可读的Streams

现在我们知道如何从 `Streams` 中读取数据，下一步是学习如何实现一个新的 `Readable` 数据流。为此，有必要通过继承 `stream.Readable` 的原型来创建一个新的类。 具体流必须提供 `_read()` 方法的实现：

```
readable._read(size)
```

`Readable` 类的内部将调用 `_read()` 方法，而该方法又将启动 使用 `push()` 填充内部缓冲区：

> 请注意，read()是Stream消费者调用的方法，而_read()是一个由Stream子类实现的方法，不能直接调用。下划线通常表示该方法为私有方法，不应该直接调用。

为了演示如何实现新的可读 `Streams` ，我们可以尝试实现一个生成随机字符串的 `Streams` 。 我们来创建一个名为 `randomStream.js` 的新模块，它将包含我们的字符串的 `generator` 的代码：

```
const stream = require('stream');
const Chance = require('chance');

const chance = new Chance();

class RandomStream extends stream.Readable {
  constructor(options) {
    super(options);
  }

  _read(size) {
    const chunk = chance.string(); //[1]
    console.log(`Pushing chunk of size: ${chunk.length}`);
    this.push(chunk, 'utf8'); //[2]
    if (chance.bool({
        likelihood: 5
      })) { //[3]
      this.push(null);
    }
  }
}

module.exports = RandomStream;
```

在文件顶部，我们将加载我们的依赖关系。除了我们正在加载一个chance的npm模块之外，没有什么特别之处，它是一个用于生成各种随机值的库，从数字到字符串到整个句子都能生成随机值。

下一步是创建一个名为 `RandomStream` 的新类，并指定 `stream.Readable` 作为其父类。 在前面的代码中，我们调用父类的构造函数来初始化其内部状态，并将收到的 `options` 参数作为输入。通过 `options` 对象传递的可能参数包括以下内容：

- 用于将 `Buffers` 转换为 `Strings` 的 `encoding` 参数（默认值为 `null` ）
- 是否启用对象模式（ `objectMode` 默认为 `false` ）
- 存储在内部 `buffer` 区中的数据的上限，一旦超过这个上限，则暂停从 `data source` 读取（ `highWaterMark` 默认为 `16KB` ）

好的，现在让我们来解释一下我们重写的 `stream.Readable` 类的 `_read()` 方法：

- 该方法使用 `chance` 生成随机字符串。
- 它将字符串 `push` 内部 `buffer` 。 请注意，由于我们 `push` 的是 `String` ，此外我们还指定了编码为 `utf8` （如果数据块只是一个二进制 `Buffer` ，则不需要）。
- 以 `5%` 的概率随机中断 `stream` 的随机字符串产生，通过 `push` `null` 到内部 `Buffer` 来表示 `EOF` ，即 `stream` 的结束。

我们还可以看到在 `_read()` 函数的输入中给出的 `size` 参数被忽略了，因为它是一个建议的参数。 我们可以简单地把所有可用的数据都 `push` 到内部的 `buffer` 中，但是如果在同一个调用中有多个推送，那么我们应该检查 `push()` 是否返回 `false` ，因为这意味着内部 `buffer` 已经达到了 `highWaterMark` 限制，我们应该停止添加更多的数据。

以上就是 `RandomStream` 模块，我们现在准备好使用它。我们来创建一个名为 `generateRandom.js` 的新模块，在这个模块中我们实例化一个新的 `RandomStream` 对象并从中提取一些数据：

```
const RandomStream = require('./randomStream');
const randomStream = new RandomStream();

randomStream.on('readable', () => {
  let chunk;
  while ((chunk = randomStream.read()) !== null) {
    console.log(`Chunk received: ${chunk.toString()}`);
  }
});
```

现在，一切都准备好了，我们尝试新的自定义的 `stream` 。 像往常一样简单地执行 `generateRandom` 模块，观察随机的字符串在屏幕上流动。

可写的Streams

一个可写的 `stream` 表示一个数据终点，在 `Node.js` 中，它使用 `stream` 模块中的 `Writable` 抽象类来实现。

写入一个stream

把一些数据放在可写入的 `stream` 中是一件简单的事情， 我们所要做的就是使用 `write()` 方法，它具有以下格式：

```
writable.write(chunk, [encoding], [callback])
```

`encoding` 参数是可选的，其在 `chunk` 是 `String` 类型时指定（默认为 `utf8` ，如果 `chunk` 是 `Buffer` ，则忽略）；当数据块被刷新到底层资源中时， `callback` 就会被调用， `callback` 参数也是可选的。

为了表示没有更多的数据将被写入 `stream` 中，我们必须使用 `end()` 方法：

```
writable.end([chunk], [encoding], [callback])
```

我们可以通过 `end()` 方法提供最后一块数据。在这种情况下， `callbak` 函数相当于为 `finish` 事件注册一个监听器，当数据块全部被写入 `stream` 中时，会触发该事件。

现在，让我们通过创建一个输出随机字符串序列的小型 `HTTP` 服务器来演示这是如何工作的：

```
const Chance = require('chance');
const chance = new Chance();

require('http').createServer((req, res) => {
  res.writeHead(200, {
    'Content-Type': 'text/plain'
  }); //[1]
  while (chance.bool({
      likelihood: 95
    })) { //[2]
    res.write(chance.string() + '\n'); //[3]
  }
  res.end('\nThe end...\n'); //[4]
  res.on('finish', () => console.log('All data was sent')); //[5]

}).listen(8080, () => console.log('Listening on http://localhost
:8080'));
```

我们创建了一个 `HTTP服务器` ，并把数据写入 `res` 对象， `res` 对象是 `http.ServerResponse` 的一个实例，也是一个可写入的 `stream` 。下面来解释上述代码发生了什么：

1. 我们首先写 `HTTP response` 的头部。请注意， `writeHead()` 不是 `Writable` 接口的一部分，实际上，这个方法是 `http.ServerResponse` 类公开的辅助方法。
2. 我们开始一个 `5%` 的概率终止的循环（进入循环体的概率为 `chance.bool()` 产生，其为 `95%` ）。
3. 在循环内部，我们写入一个随机字符串到 `stream` 。
4. 一旦我们不在循环中，我们调用 `stream` 的 `end()` ，表示没有更多 数据块将被写入。另外，我们在结束之前提供一个最终的字符串写入流中。
5. 最后，我们注册一个 `finish` 事件的监听器，当所有的数据块都被刷新到底层 `socket` 中时，这个事件将被触发。

我们可以调用这个小模块称为 `entropyServer.js` ，然后执行它。要测试这个服务器，我们可以在地址 `http：// localhost:8080` 打开一个浏览器，或者从终端使用 `curl` 命令，如下所示：

```
curl localhost:8080
```

此时，服务器应该开始向您选择的 `HTTP客户端` 发送随机字符串（请注意，某些浏览器可能会缓冲数据，并且流式传输行为可能不明显）。

Back-pressure（反压）

类似于在真实管道系统中流动的液体， `Node.js` 的 `stream` 也可能遭受瓶颈，数据写入速度可能快于 `stream` 的消耗。解决这个问题的机制包括缓冲输入数据；然而，如果数据 `stream` 没有给生产者任何反馈，我们可能会产生越来越多的数据被累积到内部缓冲区的情况，导致内存泄露的发生。

为了防止这种情况的发生，当内部 `buffer` 超过 `highWaterMark` 限制时， `writable.write()` 将返回 `false` 。可写入的 `stream` 具有 `highWaterMark` 属性，这是 `write()` 方法开始返回 `false` 的内部 `Buffer` 区大小的限制，一旦 `Buffer` 区的大小超过这个限制，表示应用程序应该停止写入。当缓冲器被清空时，会触发一个叫做 `drain` 的事件，通知再次开始写入是安全的。这种机制被称为 `back-pressure` 。

> 本节介绍的机制同样适用于可读的stream。事实上，在可读stream中也存在back-pressure，并且在_read()内调用的push()方法返回false时触发。但是，这对于stream实现者来说是一个特定的问题，所以我们将不经常处理它。

我们可以通过修改之前创建的 `entropyServer` 模块来演示可写入的 `stream` 的 `back-pressure` ：

```
const Chance = require('chance');
const chance = new Chance();

require('http').createServer((req, res) => {
  res.writeHead(200, {
    'Content-Type': 'text/plain'
  });

  function generateMore() { //[1]
    while (chance.bool({
        likelihood: 95
      })) {
      const shouldContinue = res.write(
        chance.string({
          length: (16 * 1024) - 1
        }) //[2]
      );
      if (!shouldContinue) { //[3]
        console.log('Backpressure');
        return res.once('drain', generateMore);
      }
    }
    res.end('\nThe end...\n', () => console.log('All data was se
nt'));
  }
  generateMore();
}).listen(8080, () => console.log('Listening on http://localhost
:8080'));
```

前面代码中最重要的步骤可以概括如下：

1. 我们将主逻辑封装在一个名为 `generateMore()` 的函数中。
2. 为了增加获得一些 `back-pressure` 的机会，我们将数据块的大小增加到 `16KB-1Byte` ，这非常接近默认的 `highWaterMark` 限制。
3. 在写入一大块数据之后，我们检查 `res.write()` 的返回值。 如果它返回 `false` ，这意味着内部 `buffer` 已满，我们应该停止发送更多的数据。在这种情况下，我们从函数中退出，然后新注册一个写入事件的发布者，当 `drain` 事件触发时调用 `generateMore` 。

如果我们现在尝试再次运行服务器，然后使用 `curl` 生成客户端请求，则很可能会有一些 `back-pressure` ，因为服务器以非常高的速度生成数据，速度甚至会比底层 `socket` 更快。

实现可写入的Streams

我们可以通过继承 `stream.Writable` 类来实现一个新的可写入的流，并为 `_write()` 方法提供一个实现。实现一个我们自定义的可写入的 `Streams` 类。

让我们构建一个可写入的 `stream` ，它接收对象的格式如下：

```
{
  path: <path to a file>
  content: <string or buffer>
}
```

这个类的作用是这样的：对于每一个对象，我们的 `stream` 必须将 `content` 部分保存到在给定路径中创建的文件中。 我们可以立即看到，我们 `stream` 的输入是对象，而不是 `Strings` 或 `Buffers` ，这意味着我们的 `stream` 必须以对象模式工作。

调用模块 `toFileStream.js` ：

```
const stream = require('stream');
const fs = require('fs');
const path = require('path');
const mkdirp = require('mkdirp');

class ToFileStream extends stream.Writable {
  constructor() {
    super({
      objectMode: true
    });
  }

  _write(chunk, encoding, callback) {
    mkdirp(path.dirname(chunk.path), err => {
      if (err) {
        return callback(err);
      }
      fs.writeFile(chunk.path, chunk.content, callback);
    });
  }
}
module.exports = ToFileStream;
```

作为第一步，我们加载所有我们所需要的依赖包。注意，我们需要模块 `mkdirp` ，正如你应该从前几章中所知道的，它应该使用 `npm` 安装。

我们创建了一个新类，它从 `stream.Writable` 扩展而来。

我们不得不调用父构造函数来初始化其内部状态；我们还提供了一个 `option` 对象作为参数，用于指定流在对象模式下工作（ `objectMode：true` ）。 `stream.Writable` 接受的其他选项如下：

- `highWaterMark` （默认值是 `16KB` ）：控制 `back-pressure` 的上限。
- `decodeStrings` （默认为 `true` ）：在字符串传递给 `_write()` 方法之前，将字符串自动解码为二进制 `buffer` 区。 在对象模式下这个参数被忽略。

最后，我们为 `_write()` 方法提供了一个实现。正如你所看到的，这个方法接受一个数据块，一个编码方式（只有在二进制模式下， `stream` 选项 `decodeStrings` 设置为 `false` 时才有意义）。

另外，该方法接受一个回调函数，该函数在操作完成时需要调用；而不必要传递操作的结果，但是如果需要的话，我们仍然可以传递一个 `error` 对象，这将导致 `stream` 触发 `error` 事件。

现在，为了尝试我们刚刚构建的 `stream` ，我们可以创建一个名为 `writeToFile.js` 的新模块，并对该流执行一些写操作：

```
const ToFileStream = require('./toFileStream.js');
const tfs = new ToFileStream();

tfs.write({path: "file1.txt", content: "Hello"});
tfs.write({path: "file2.txt", content: "Node.js"});
tfs.write({path: "file3.txt", content: "Streams"});
tfs.end(() => console.log("All files created"));
```

有了这个，我们创建并使用了我们的第一个自定义的可写入流。 像往常一样运行新模块来检查其输出；你会看到执行后会创建三个新文件。

双重的Streams

双重的 `stream` 既是可读的，也可写的。 当我们想描述一个既是数据源又是数据终点的实体时（例如 `socket` ），这就显得十分有用了。 双工流继承 `stream.Readable` 和 `stream.Writable` 的方法，所以它对我们来说并不新鲜。这意味着我们可以 `read()` 或 `write()` 数据，或者可以监听 `readable` 和 `drain` 事件。

要创建一个自定义的双重 `stream` ，我们必须为 `_read()` 和 `_write()` 提供一个实现。传递给 `Duplex()` 构造函数的 `options` 对象在内部被转发给 `Readable` 和 `Writable` 的构造函数。 `options` 参数的内容与前面讨论的相同， `options` 增加了一个名为 `allowHalfOpen` 值（默认为 `true` ），如果设置为 `false` ，则会导致只要 `stream` 的一方（ `Readable` 和 `Writable` ）结束， `stream` 就结束了。

> 为了使双重的stream在一方以对象模式工作，而在另一方以二进制模式工作，我们需要在流构造器中手动设置以下属性：

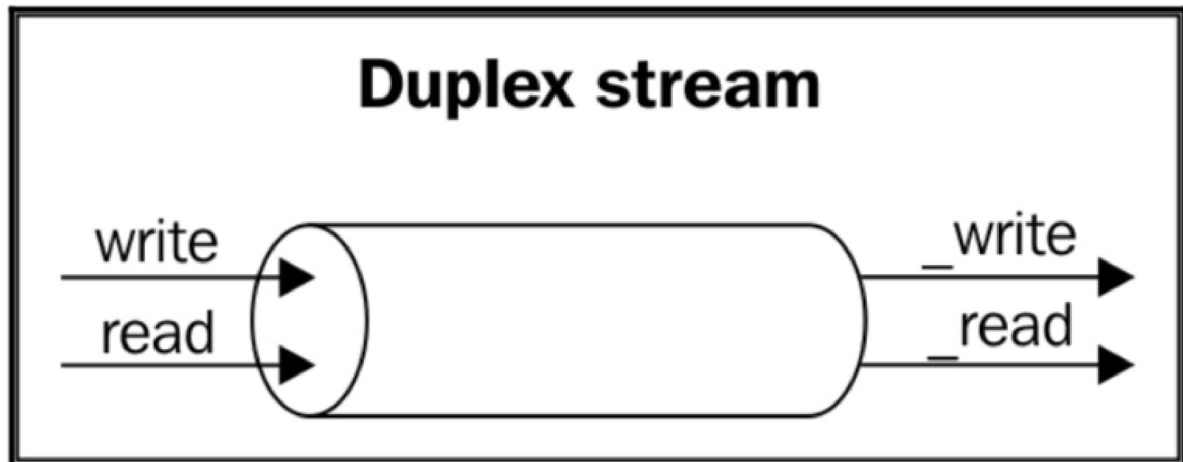
```
this._writableState.objectMode
this._readableState.objectMode
```

转换的Streams

转换的 `Streams` 是专门设计用于处理数据转换的一种特殊类型的双重 `Streams` 。

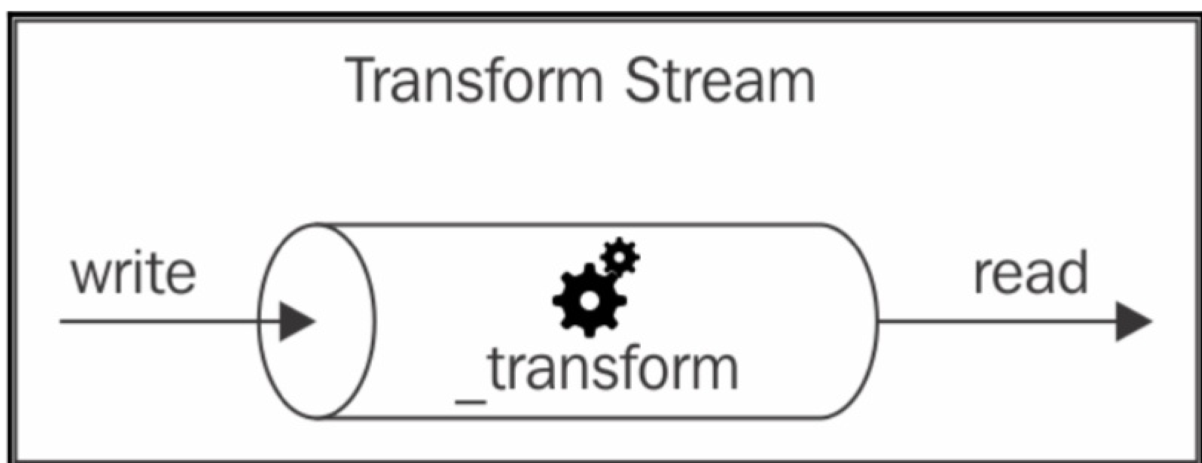
在一个简单的双重 `Streams` 中，从 `stream` 中读取的数据和写入到其中的数据之间没有直接的关系（至少 `stream` 是不可知的）。想想一个 `TCP socket` ，它只是向远程节点发送数据和从远程节点接收数据。 `TCP socket` 自身没有意识到输入和输出之间有任何关系。

下图说明了双重 `Streams` 中的数据流：



另一方面，转换的 `Streams` 对从可写入端接收到的每个数据块应用某种转换，然后在其可读端使转换的数据可用。

下图显示了数据如何在转换的 `Streams` 中流动：



从外面看，转换的 `Streams` 的接口与双重 `Streams` 的接口完全相同。但是，当我们想要构建一个新的双重 `Streams` 时，我们必须提供 `_read()` 和 `_write()` 方法，而为了实现一个新的变换流，我们必须填写另一对方法： `_transform()` 和 `_flush()` ）。

我们来演示如何用一个例子来创建一个新的转换的 `Streams` 。

实现转换的Streams

我们来实现一个转换的 `Streams` ，它将替换给定所有出现的字符串。 要做到这一点，我们必须创建一个名为 `replaceStream.js` 的新模块。 让我们直接看怎么实现它：

```
const stream = require('stream');
const util = require('util');

class ReplaceStream extends stream.Transform {
  constructor(searchString, replaceString) {
    super();
    this.searchString = searchString;
    this.replaceString = replaceString;
    this.tailPiece = '';
  }

  _transform(chunk, encoding, callback) {
    const pieces = (this.tailPiece + chunk)         //[1]
      .split(this.searchString);
    const lastPiece = pieces[pieces.length - 1];
    const tailPieceLen = this.searchString.length - 1;

    this.tailPiece = lastPiece.slice(-tailPieceLen);     //[2]
    pieces[pieces.length - 1] = lastPiece.slice(0,-tailPieceLen)
;

    this.push(pieces.join(this.replaceString));       //[3]
    callback();
  }

  _flush(callback) {
    this.push(this.tailPiece);
    callback();
  }
}

module.exports = ReplaceStream;
```

与往常一样，我们将从其依赖项开始构建模块。这次我们没有使用第三方模块。

然后我们创建了一个从 `stream.Transform` 基类继承的新类。该类的构造函数接受两个参数： `searchString` 和 `replaceString` 。 正如你所想象的那样，它们允许我们定义要匹配的文本以及用作替换的字符串。 我们还初始化一个将由 `_transform()` 方法使用的 `tailPiece` 内部变量。

现在，我们来分析一下 `_transform()` 方法，它是我们新类的核心。 `_transform()` 方法与可写入的 `stream` 的 `_write()` 方法具有几乎相同的格式，但不是将数据写入底层资源，而是使用 `this.push()` 将其推入内

部 `buffer` ，这与我们会在可读流的 `_read()` 方法中执行。这显示了转换的 `Streams` 的双方如何实际连接。

`ReplaceStream` 的 `_transform()` 方法实现了我们这个新类的核心。正常情况下，搜索和替换 `buffer` 区中的字符串是一件容易的事情；但是，当数据流式传输时，情况则完全不同，可能的匹配可能分布在多个数据块中。代码后面的程序可以解释如下：

1. 我们的算法使用 `searchString` 函数作为分隔符来分割块。
2. 然后，它取出分隔后生成的数组的最后一项 `lastPiece` ，并提取其最后一个字符 `searchString.length - 1` 。结果被保存到 `tailPiece` 变量中，它将会被作为下一个数据块的前缀。
3. 最后，所有从 `split()` 得到的片段用 `replaceString` 作为分隔符连接在一起，并推入内部 `buffer` 区。

当 `stream` 结束时，我们可能仍然有最后一个 `tailPiece` 变量没有被压入内部缓冲区。这正是 `_flush()` 方法的用途；它在 `stream` 结束之前被调用，并且这是我们最终有机会完成流或者在完全结束流之前推送任何剩余数据的地方。

`_flush()` 方法只需要一个回调函数作为参数，当所有的操作完成后，我们必须确保调用这个回调函数。完成了这个，我们已经完成了我们的 `ReplaceStream` 类。

现在，是时候尝试新的 `stream` 。我们可以创建另一个名为 `replaceStreamTest.js` 的模块来写入一些数据，然后读取转换的结果：

```
const ReplaceStream = require('./replaceStream');

const rs = new ReplaceStream('World', 'Node.js');
rs.on('data', chunk => console.log(chunk.toString()));

rs.write('Hello W');
rs.write('orld!');
rs.end();
```

为了使得这个例子更复杂一些，我们把搜索词分布在两个不同的数据块上；然后，使用 `flowing` 模式，我们从同一个 `stream` 中读取数据，记录每个已转换的块。运行前面的程序应该产生以下输出：

```
Hel
lo Node.js
!
```

> 有一个值得提及是，第五种类型的stream：stream.PassThrough。 与我们介绍的其他流类不同，PassThrough不是抽象的，可以直接实例化，而不需要实现任何方法。实际上，这是一个可转换的stream，它可以输出每个数据块，而不需要进行任何转换。

使用管道连接Streams

`Unix` 管道的概念是由 `Douglas Mcllroy` 发明的；这使程序的输出能够连接到下一个的输入。看看下面的命令：

```
echo Hello World! | sed s/World/Node.js/g
```

在前面的命令中， `echo` 会将 `Hello World!` 写入标准输出，然后被重定向到 `sed` 命令的标准输入（因为有管道操作符 `|` ）。然后 `sed` 用 `Node.js` 替换任何 `World` ，并将结果打印到它的标准输出（这次是控制台）。

以类似的方式，可以使用可读的 `Streams` 的 `pipe()` 方法将 `Node.js` 的 `Streams` 连接在一起，它具有以下接口：

```
readable.pipe(writable, [options])
```

非常直观地， `pipe()` 方法将从可读的 `Streams` 中发出的数据抽取到所提供的可写入的 `Streams` 中。另外，当可读的 `Streams` 发出 `end` 事件（除非我们指定 `{end：false}` 作为 `options` ）时，可写入的 `Streams` 将自动结束。 `pipe()` 方法返回作为参数传递的可写入的 `Streams` ，如果这样的 `stream` 也是可读的（例如双重或可转换的 `Streams` ），则允许我们创建链式调用。

将两个 `Streams` 连接到一起时，则允许数据自动流向可写入的 `Streams` ，所以不需要调用 `read()` 或 `write()` 方法；但最重要的是不需要控制 `back-pressure` ，因为它会自动处理。

举个简单的例子（将会有大量的例子），我们可以创建一个名为 `replace.js` 的新模块，它接受来自标准输入的文本流，应用替换转换，然后将数据返回到标准输出：

```
const ReplaceStream = require('./replaceStream');
process.stdin
  .pipe(new ReplaceStream(process.argv[2], process.argv[3]))
  .pipe(process.stdout);
```

上述程序将来自标准输入的数据传送到 `ReplaceStream` ，然后返回到标准输出。现在，为了实践这个小应用程序，我们可以利用 `Unix` 管道将一些数据重定向到它的标准输入，如下所示：

```
echo Hello World! | node replace World Node.js
```

运行上述程序，会输出如下结果：

```
Hello Node.js
```

这个简单的例子演示了 `Streams` （特别是文本 `Streams` ）是一个通用接口，管道几乎是构成和连接所有这些接口的通用方式。

> `error` 事件不会通过管道自动传播。举个例子，看如下代码片段：

```
stream1
  .pipe(stream2)
  .on('error', function() {});
```

> 在前面的链式调用中，我们将只捕获来自 `stream2` 的错误，这是由于我们给其添加了 `erorr` 事件侦听器。这意味着，如果我们想捕获从 `stream1` 生成的任何错误，我们必须直接附加另一个错误侦听器。 稍后我们将看到一种可以实现共同错误捕获的另一种模式（合并 `Streams` ）。 此外，我们应该注意到，如果目标 `Streams` （读取的 `Streams` ）发出错误，它将会对源 `Streams` 通知一个 `error` ，之后导致管道的中断。

Streams如何通过管道

到目前为止，我们创建自定义 `Streams` 的方式并不完全遵循 `Node` 定义的模式；实际上，从 `stream` 基类继承是违反 `small surface area` 的，并需要一些示例代码。 这并不意味着 `Streams` 设计得不好，实际上，我们不应该忘记，因为 `Streams` 是 `Node.js` 核心的一部分，所以它们必须尽可能地灵活，广泛拓展 `Streams` 以致于用户级模块能够将它们充分运用。

然而，大多数情况下，我们并不需要原型继承可以给予的所有权力和可扩展性，但通常我们想要的仅仅是定义新 `Streams` 的一种快速开发的模式。 `Node.js` 社区当然也为此创建了一个解决方案。 一个完美的例子是through2，一个使得我们可以简单地创建转换的 `Streams` 的小型库。 通过 `through2` ，我们可以通过调用一个简单的函数来创建一个新的可转换的 `Streams` ：

```
const transform = through2([options], [_transform], [_flush]);
```

类似的，from2也允许我们像下面这样创建一个可读的 `Streams` ：

```
const readable = from2([options], _read);
```

接下来，我们将在本章其余部分展示它们的用法，那时，我们会清楚使用这些小型库的好处。

> through和from是基于 `Stream1` 规范的顶层库。

基于Streams的异步控制流

通过我们已经介绍的例子，应该清楚的是， `Streams` 不仅可以用来处理 `I / O` ，而且可以用作处理任何类型数据的优雅编程模式。 但优点并不止这些；还可以利用 `Streams` 来实现异步控制流，在本节将会看到。

顺序执行

默认情况下， `Streams` 将按顺序处理数据；例如，转换的 `Streams` 的 `_transform()` 函数在前一个数据块执行 `callback()` 之后才会进行下一块数据块的调用。这是 `Streams` 的一个重要属性，按正确顺序处理每个数据块至关重要，但是也可以利用这一属性将 `Streams` 实现优雅的传统控制流模式。

代码总是比太多的解释要好得多，所以让我们来演示一下如何使用流来按顺序执行异步任务的例子。让我们创建一个函数来连接一组接收到的文件作为输入，确保遵守提供的顺序。我们创建一个名为 `concatFiles.js` 的新模块，并从其依赖开始：

```
const fromArray = require('from2-array');
const through = require('through2');
const fs = require('fs');
```

我们将使用 `through2` 来简化转换的 `Streams` 的创建，并使用 `from2-array` 从一个对象数组中创建可读的 `Streams` 。 接下来，我们可以定义 `concatFiles()` 函数：

```
function concatFiles(destination, files, callback) {
  const destStream = fs.createWriteStream(destination);
  fromArray.obj(files)                    //[1]
    .pipe(through.obj((file, enc, done) => {   //[2]
      const src = fs.createReadStream(file);
      src.pipe(destStream, {end: false});
      src.on('end', done); //[3]
    }))
    .on('finish', () => {          //[4]
      destStream.end();
      callback();
    });
}

module.exports = concatFiles;
```

前面的函数通过将 `files` 数组转换为 `Streams` 来实现对 `files` 数组的顺序迭代。 该函数所遵循的程序解释如下：

1. 首先，我们使用 `from2-array` 从 `files` 数组创建一个可读的 `Streams` 。
2. 接下来，我们使用 `through` 来创建一个转换的 `Streams` 来处理序列中的每个文件。对于每个文件，我们创建一个可读的 `Streams` ，并通过管道将其输入到表示输出文件的 `destStream` 中。 在源文件完成读取后，通过在 `pipe()` 方法的第二个参数中指定 `{end：false}` ，我们确保不关闭 `destStream` 。
3. 当源文件的所有内容都被传送到 `destStream` 时，我们调用 `through.obj` 公开的 `done` 函数来传递当前处理已经完成，在我们的情况下这是需要触发处理下一个文件。
4. 所有文件处理完后， `finish` 事件被触发。我们最后可以结束 `destStream` 并调用 `concatFiles()` 的 `callback()` 函数，这个函数表示整个操作的完成。

我们现在可以尝试使用我们刚刚创建的小模块。让我们创建一个名为 `concat.js` 的新文件来完成一个示例：

```
const concatFiles = require('./concatFiles');

concatFiles(process.argv[2], process.argv.slice(3), () => {
  console.log('Files concatenated successfully');
});
```

我们现在可以运行上述程序，将目标文件作为第一个命令行参数，接着是要连接的文件列表，例如：

```
node concat allTogether.txt file1.txt file2.txt
```

执行这一条命令，会创建一个名为 `allTogether.txt` 的新文件，其中按顺序保存 `file1.txt` 和 `file2.txt` 的内容。

使用 `concatFiles()` 函数，我们能够仅使用 `Streams` 实现异步操作的顺序执行。正如我们在 `Chapter3 Asynchronous Control Flow Patters with Callbacks` 中看到的那样，如果使用纯 `JavaScript` 实现，或者使用 `async` 等外部库，则需要使用或实现迭代器。我们现在提供了另外一个可以达到同样效果的方法，正如我们所看到的，它的实现方式非常优雅且可读性高。

> 模式：使用Streams或Streams的组合，可以轻松地按顺序遍历一组异步任务。

无序并行执行

我们刚刚看到 `Streams` 按顺序处理每个数据块，但有时这可能并不能这么做，因为这样并没有充分利用 `Node.js` 的并发性。如果我们必须对每个数据块执行一个缓慢的异步操作，那么并行化执行这一组异步任务完全是有必要的。当然，只有在

每个数据块之间没有关系的情况下才能应用这种模式，这些数据块可能经常发生在对象模式的 `Streams` 中，但是对于二进制模式的 `Streams` 很少使用无序的并行执行。

> 注意：当处理数据的顺序很重要时，不能使用无序并行执行的Streams。

为了并行化一个可转换的 `Streams` 的执行，我们可以运用 `Chapter3 Asynchronous Control Flow Patters with Callbacks` 所讲到的无序并行执行的相同模式，然后做出一些改变使它们适用于 `Streams` 。让我们看看这是如何更改的。

实现一个无序并行的**Streams**

让我们用一个例子直接说明：我们创建一个叫做 `parallelStream.js` 的模块，然后自定义一个普通的可转换的 `Streams` ，然后给出一系列可转换流的方法：

```
const stream = require('stream');

class ParallelStream extends stream.Transform {
  constructor(userTransform) {
    super({objectMode: true});
    this.userTransform = userTransform;
    this.running = 0;
    this.terminateCallback = null;
  }

  _transform(chunk, enc, done) {
    this.running++;
    this.userTransform(chunk, enc, this._onComplete.bind(this),
this.push.bind(this));
    done();
  }

  _flush(done) {
    if(this.running > 0) {
      this.terminateCallback = done;
    } else {
      done();
    }
  }

  _onComplete(err) {
    this.running--;
    if(err) {
      return this.emit('error', err);
    }
    if(this.running === 0) {
      this.terminateCallback && this.terminateCallback();
    }
  }
}

module.exports = ParallelStream;
```

我们来分析一下这个新的自定义的类。正如你所看到的一样，构造函数接受一个 `userTransform()` 函数作为参数，然后将其另存为一个实例变量；我们也调用父构造函数，并且我们默认启用对象模式。

接下来，来看 `_transform()` 方法，在这个方法中，我们执行 `userTransform()` 函数，然后增加当前正在运行的任务个数; 最后，我们通过调用 `done()` 来通知当前转换步骤已经完成。 `_transform()` 方法展示了如何并行处理另一项任务。我们不用等待 `userTransform()` 方法执行完毕再调用 `done()` 。相反，我们立即执行 `done()` 方法。另一方面，我们提供了一个特殊的回调函数给 `userTransform()` 方法，这就是 `this._onComplete()` 方法；以便我们在 `userTransform()` 完成的时候收到通知。

在 `Streams` 终止之前，会调用 `_flush()` 方法，所以如果仍有任务正在运行，我们可以通过不立即调用 `done()` 回调函数来延迟 `finish` 事件的触发。相反，我们将其分配给 `this.terminateCallback` 变量。为了理解 `Streams` 如何正确终止，来看 `_onComplete()` 方法。

在每组异步任务最终完成时， `_onComplete()` 方法会被调用。首先，它会检查是否有任务正在运行，如果没有，则调用 `this.terminateCallback()` 函数，这将导致 `Streams` 结束，触发 `_flush()` 方法的 `finish` 事件。

利用刚刚构建的 `ParallelStream` 类可以轻松地创建一个无序并行执行的可转换的 `Streams` 实例，但是有个注意：它不会保留项目接收的顺序。实际上，异步操作可以在任何时候都有可能完成并推送数据，而跟它们开始的时刻并没有必然的联系。因此我们知道，对于二进制模式的 `Streams` 并不适用，因为二进制的 `Streams` 对顺序要求较高。

实现一个**URL**监控应用程序

现在，让我们使用 `ParallelStream` 模块实现一个具体的例子。让我们想象以下我们想要构建一个简单的服务来监控一个大 `URL` 列表的状态，让我们想象以下，所有的这些 `URL` 包含在一个单独的文件中，并且每一个 `URL` 占据一个空行。

`Streams` 能够为这个场景提供一个高效且优雅的解决方案。特别是当我们使用我们刚刚写的 `ParallelStream` 类来无序地审核这些 `URL` 。

接下来，让我们创建一个简单的放在 `checkUrls.js` 模块的应用程序。

```
const fs = require('fs');
const split = require('split');
const request = require('request');
const ParallelStream = require('./parallelStream');

fs.createReadStream(process.argv[2])         //[1]
  .pipe(split())                             //[2]
  .pipe(new ParallelStream((url, enc, done, push) => {     //[3]
    if(!url) return done();
    request.head(url, (err, response) => {
      push(url + ' is ' + (err ? 'down' : 'up') + '\n');
      done();
    });
  }))
  .pipe(fs.createWriteStream('results.txt'))   //[4]
  .on('finish', () => console.log('All urls were checked'))
;
```

正如我们所看到的，通过流，我们的代码看起来非常优雅，直观。 让我们看看它是如何工作的：

1. 首先，我们通过给定的文件参数创建一个可读的 `Streams` ，便于接下来读取文件。

2. 我们通过split将输入的文件的 `Streams` 的内容输出一个可转换的 `Streams` 到管道中，并且分隔了数据块的每一行。
3. 然后，是时候使用我们的 `ParallelStream` 来检查 `URL` 了，我们发送一个 `HEAD` 请求然后等待请求的 `response` 。当请求返回时，我们把请求的结果 `push` 到 `stream` 中。
4. 最后，通过管道把结果保存到 `results.txt` 文件中。

```
node checkUrls urlList.txt
```

这里的文件 `urlList.txt` 包含一组 `URL` ，例如：

- `http://www.mariocasciaro.me/`
- `http://loige.co/`
- `http://thiswillbedownforsure.com/`

当应用执行完成后，我们可以看到一个文件 `results.txt` 被创建，里面包含有操作的结果，例如：

- `http://thiswillbedownforsure.com is down`
- `http://loige.co is up`
- `http://www.mariocasciaro.me is up`

输出的结果的顺序很有可能与输入文件中指定 `URL` 的顺序不同。这是 `Streams` 无序并行执行任务的明显特征。

> 出于好奇，我们可能想尝试用一个正常的through2流替换ParallelStream，并比较两者的行为和性能（你可能想这样做的一个练习）。我们将会看到，使用through2的方式会比较慢，因为每个URL都将按顺序进行检查，而且文件results.txt中结果的顺序也会被保留。

无序限制并行执行

如果运行包含数千或数百万个URL的文件的 `checkUrls` 应用程序，我们肯定会遇到麻烦。我们的应用程序将同时创建不受控制的连接数量，并行发送大量数据，并可能破坏应用程序的稳定性和整个系统的可用性。我们已经知道，控制负载的无序限制并行执行是一个极好的解决方案。

让我们通过创建一个 `limitedParallelStream.js` 模块来看看它是如何工作的，这个模块是改编自上一节中创建的 `parallelStream.js` 模块。

让我们看看它的构造函数：

```
class LimitedParallelStream extends stream.Transform {
  constructor(concurrency, userTransform) {
    super({objectMode: true});
    this.concurrency = concurrency;
    this.userTransform = userTransform;
    this.running = 0;
    this.terminateCallback = null;
    this.continueCallback = null;
  }
// ...
}
```

我们需要一个 `concurrency` 变量作为输入来限制并发量，这次我们要保存两个回调函数， `continueCallback` 用于任何挂起的 `_transform` 方法， `terminateCallback` 用于_flush方法的回调。 接下来看 `_transform()` 方法：

```
_transform(chunk, enc, done) {
  this.running++;
  this.userTransform(chunk, enc,  this.push.bind(this), this._on
Complete.bind(this));
  if(this.running < this.concurrency) {
    done();
  } else {
    this.continueCallback = done;
  }
}
```

这次在 `_transform()` 方法中，我们必须在调用 `done()` 之前检查是否达到了最大并行数量的限制，如果没有达到了限制，才能触发下一个项目的处理。如果我们已经达到到最大并行数量的限制，我们可以简单地将 `done()` 回调保存到 `continueCallback` 变量中，以便在任务完成后立即调用它。

`_flush()` 方法与 `ParallelStream` 类保持完全一样，所以我们直接转到实现 `_onComplete()` 方法：

```
_onComplete(err) {
  this.running--;
  if(err) {
    return this.emit('error', err);
  }
  const tmpCallback = this.continueCallback;
  this.continueCallback = null;
  tmpCallback && tmpCallback();
  if(this.running === 0) {
    this.terminateCallback && this.terminateCallback();
  }
}
```

每当任务完成，我们调用任何已保存的 `continueCallback()` 将导致 `stream` 解锁，触发下一个项目的处理。

这就是 `limitedParallelStream` 模块。 我们现在可以在 `checkUrls` 模块中使用它来代替 `parallelStream` ，并且将我们的任务的并发限制在我们设置的值上。

顺序并行执行

我们以前创建的并行 `Streams` 可能会使得数据的顺序混乱，但是在某些情况下这是不可接受的。有时，实际上，有那种需要每个数据块都以接收到的相同顺序发出的业务场景。我们仍然可以并行运行 `transform` 函数。我们所要做的就是对每个任务发出的数据进行排序，使其遵循与接收数据相同的顺序。

这种技术涉及使用 `buffer` ，在每个正在运行的任务发出时重新排序块。为简洁起见，我们不打算提供这样一个 `stream` 的实现，因为这本书的范围是相当冗长的；我们要做的就是重用为了这个特定目的而构建的 `npm` 上的一个可用包，例如 through2-parallel。

我们可以通过修改现有的 `checkUrls` 模块来快速检查一个有序的并行执行的行为。 假设我们希望我们的结果按照与输入文件中的 `URL` 相同的顺序编写。 我们可以使用通过 `through2-parallel` 来实现：

```
const fs = require('fs');
const split = require('split');
const request = require('request');
const throughParallel = require('through2-parallel');

fs.createReadStream(process.argv[2])
  .pipe(split())
  .pipe(throughParallel.obj({concurrency: 2}, function (url, enc
, done) {
    if(!url) return done();
    request.head(url, (err, response) => {
      this.push(url + ' is ' + (err ? 'down' : 'up') + '\n');
      done();
    });
  }))
  .pipe(fs.createWriteStream('results.txt'))
  .on('finish', () => console.log('All urls were checked'))
;
```

正如我们所看到的， `through2-parallel` 的接口与 `through2` 的接口非常相似；唯一的不同是在 `through2-parallel` 还可以为我们提供的 `transform` 函数指定一个并发限制。如果我们尝试运行这个新版本的 `checkUrls` ，我们会看到 `results.txt` 文件列出结果的顺序与输入文件中URLs的出现顺序是一样的。

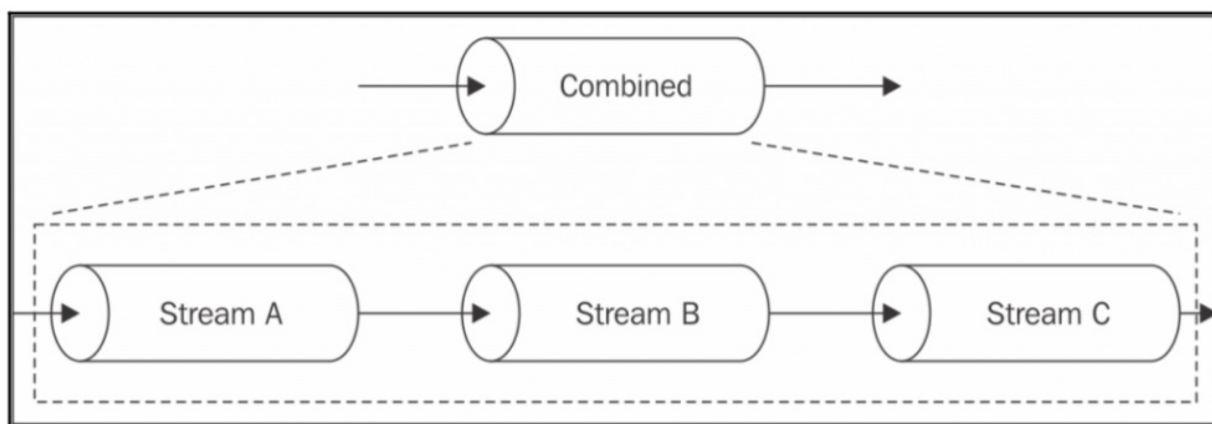
通过这个，我们总结了使用 `Streams` 实现异步控制流的分析；接下来，我们研究管道模式。

管道模式

就像在现实生活中一样， `Node.js` 的 `Streams` 也可以按照不同的模式进行管道连接。事实上，我们可以将两个不同的 `Streams` 合并成一个 `Streams` ，将一个 `Streams` 分成两个或更多的管道，或者根据条件重定向流。 在本节中，我们将探讨可应用于 `Node.js` 的 `Streams` 最重要的管道技术。

组合的Streams

在本章中，我们强调 `Streams` 提供了一个简单的基础结构来模块化和重用我们的代码，但是却漏掉了一个重要的部分：如果我们想要模块化和重用整个流水线？如果我们想要合并多个 `Streams` ，使它们看起来像外部的 `Streams` ，那该怎么办？下图显示了这是什么意思：



从上图中，我们看到了如何组合几个流的了：

- 当我们写入组合的 `Streams` 的时候，实际上我们是写入组合的 `Streams` 的第一个单元，即 `StreamA` 。
- 当我们从组合的 `Streams` 中读取信息时，实际上我们从组合的 `Streams` 的最后一个单元中读取。

一个组合的 `Streams` 通常是一个多重的 `Streams` ，通过连接第一个单元的写入端和连接最后一个单元的读取端。

> 要从两个不同的Streams（一个可读的Streams和一个可写入的Streams）中创建一个多重的Streams，我们可以使用一个npm模块，例如duplexer2。

但上述这么做并不完整。实际上，组合的 `Streams` 还应该做到捕获到管道中任意一段 `Streams` 单元产生的错误。我们已经说过，任何错误都不会自动传播到管道中。 所以，我们必须有适当的错误管理，我们将不得不显式附加一个错误监听器到每个 `Streams` 。但是，组合的 `Streams` 实际上是一个黑盒，这意味着我们无法访问管道中间的任何单元，所以对于管道中任意单元的异常捕获，组合的 `Streams` 也充当聚合器的角色。

总而言之，组合的 `Streams` 具有两个主要优点：

- 管道内部是一个黑盒，对使用者不可见。
- 简化了错误管理，因为我们不必为管道中的每个单元附加一个错误侦听器，而只需要给组合的 `Streams` 自身附加上就可以了。

组合的 `Streams` 是一个非常通用和普遍的做法，所以如果我们没有任何特殊的需要，我们可能只想重用现有的解决方案，如multipipe或combine-stream。

实现一个组合的Streams

为了说明一个简单的例子，我们来考虑下面两个组合的 `Streams` 的情况：

- 压缩和加密数据
- 解压和解密数据

使用诸如 `multipipe` 之类的库，我们可以通过组合一些核心库中已有的 `Streams` （文件 `combinedStreams.js` ）来轻松地构建组合的 `Streams` ：

```
const zlib = require('zlib');
const crypto = require('crypto');
const combine = require('multipipe');
module.exports.compressAndEncrypt = password => {
  return combine(
    zlib.createGzip(),
    crypto.createCipher('aes192', password)
  );
};
module.exports.decryptAndDecompress = password => {
  return combine(
    crypto.createDecipher('aes192', password),
    zlib.createGunzip()
  );
};
```

例如，我们现在可以使用这些组合的数据流，如同黑盒，这些对我们均是不可见的，可以创建一个小型应用程序，通过压缩和加密来归档文件。 让我们在一个名为 `archive.js` 的新模块中做这件事：

```
const fs = require('fs');
const compressAndEncryptStream = require('./combinedStreams').co
mpressAndEncrypt;
fs.createReadStream(process.argv[3])
  .pipe(compressAndEncryptStream(process.argv[2]))
  .pipe(fs.createWriteStream(process.argv[3] + ".gz.enc"));
```

我们可以通过从我们创建的流水线中构建一个组合的 `Stream` 来进一步改进前面的代码，但这次并不只是为了获得对外不可见的黑盒，而是为了进行异常捕获。 实际上，正如我们已经提到过的那样，写下如下的代码只会捕获最后一个 `Stream` 单元发出的错误：

```
fs.createReadStream(process.argv[3])
  .pipe(compressAndEncryptStream(process.argv[2]))
  .pipe(fs.createWriteStream(process.argv[3] + ".gz.enc"))
  .on('error', function(err) {
    // 只会捕获最后一个单元的错误
    console.log(err);
  });
```

但是，通过把所有的 `Streams` 结合在一起，我们可以优雅地解决这个问题。重构后的 `archive.js` 如下：

```
const combine = require('multipipe');
   const fs = require('fs');
   const compressAndEncryptStream =
     require('./combinedStreams').compressAndEncrypt;
   combine(
     fs.createReadStream(process.argv[3])
     .pipe(compressAndEncryptStream(process.argv[2]))
     .pipe(fs.createWriteStream(process.argv[3] + ".gz.enc"))
   ).on('error', err => {
     // 使用组合的Stream可以捕获任意位置的错误
     console.log(err);
   });
```

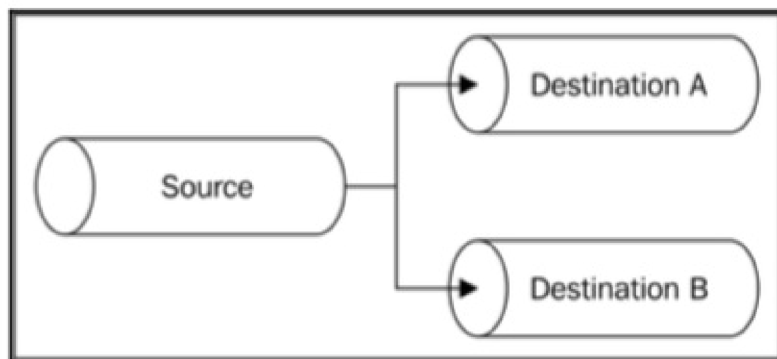
正如我们所看到的，我们现在可以将一个错误侦听器直接附加到组合的 `Streams` ，它将接收任何内部流发出的任何 `error` 事件。 现在，要运行 `archive` 模块，只需在命令行参数中指定 `password` 和 `file` 参数，即压缩模块的参数：

```
node archive mypassword /path/to/a/file.text
```

通过这个例子，我们已经清楚地证明了组合的 `Stream` 是多么重要; 从一个方面来说，它允许我们创建流的可重用组合，从另一方面来说，它简化了管道的错误管理。

分开的 `Streams`

我们可以通过将单个可读的 `Stream` 管道化为多个可写入的 `Stream` 来执行 `Stream` 的分支。当我们想要将相同的数据发送到不同的目的地时，这便体现其作用了，例如，两个不同的套接字或两个不同的文件。当我们想要对相同的数据执行不同的转换时，或者当我们想要根据一些标准拆分数据时，也可以使用它。如图所示：



在 `Node.js` 中分开的 `Stream` 是一件小事。举例说明。

实现一个多重校验和的生成器

让我们创建一个输出给定文件的 `sha1` 和 `md5` 散列的小工具。我们来调用这个新模块 `generateHashes.js` ，看如下的代码：

```
const fs = require('fs');
const crypto = require('crypto');
const sha1Stream = crypto.createHash('sha1');
sha1Stream.setEncoding('base64');
const md5Stream = crypto.createHash('md5');
md5Stream.setEncoding('base64');
```

目前为止没什么特别的 该模块的下一个部分实际上是我们将从文件创建一个可读的 `Stream` ，并将其分叉到两个不同的流，以获得另外两个文件，其中一个包含 `sha1` 散列，另一个包含 `md5` 校验和：

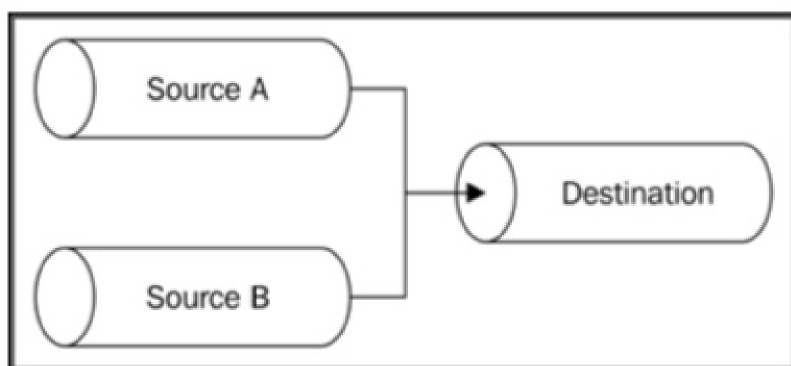
```
const inputFile = process.argv[2];
const inputStream = fs.createReadStream(inputFile);
inputStream
  .pipe(sha1Stream)
  .pipe(fs.createWriteStream(inputFile + '.sha1'));
inputStream
  .pipe(md5Stream)
  .pipe(fs.createWriteStream(inputFile + '.md5'));
```

这很简单： `inputStream` 变量通过管道一边输入到 `sha1Stream` ，另一边输入到 `md5Stream` 。但是要注意：

- 当 `inputStream` 结束时， `md5Stream` 和 `sha1Stream` 会自动结束，除非当调用 `pipe()` 时指定了 `end` 选项为 `false` 。
- `Stream` 的两个分支会接受相同的数据块，因此当对数据执行一些副作用的操作时我们必须非常谨慎，因为那样会影响另外一个分支。
- 黑盒外会产生背压，来自 `inputStream` 的数据流的流速会根据接收最慢的分支的流速作出调整。

合并的 `Streams`

合并与分开相对，通过把一组可读的 `Streams` 合并到一个单独的可写的 `Stream` 里，如图所示：



将多个 `Streams` 合并为一个通常是一个简单的操作; 然而，我们必须注意我们处理 `end` 事件的方式，因为使用自动结束选项的管道系统会在一个源结束时立即结束目标流。 这通常会导致错误，因为其他还未结束的源将继续写入已终止的 `Stream` 。 解决此问题的方法是在将多个源传输到单个目标时使用选项 `{end：false}` ，并且只有在所有源完成读取后才在目标 `Stream` 上调用 `end()` 。

用多个源文件压缩为一个压缩包

举一个简单的例子，我们来实现一个小程序，它根据两个不同目录的内容创建一个压缩包。 为此，我们将介绍两个新的 `npm` 模块：

- tar用来创建压缩包
- fstream从文件系统文件创建对象streams的库

我们创建一个新模块 `mergeTar.js` ，如下开始初始化：

```
var tar = require('tar');
var fstream = require('fstream');
var path = require('path');
var destination = path.resolve(process.argv[2]);
var sourceA = path.resolve(process.argv[3]);
var sourceB = path.resolve(process.argv[4]);
```

在前面的代码中，我们只加载全部依赖包和初始化包含目标文件和两个源目录(`sourceA` 和 `sourceB`)的变量。

接下来，我们创建 `tar` 的 `Stream` 并通过管道输出到一个可写入的 `Stream` ：

```
const pack = tar.Pack();
pack.pipe(fstream.Writer(destination));
```

现在，我们开始初始化源 `Stream`

```
let endCount = 0;

function onEnd() {
  if (++endCount === 2) {
    pack.end();
  }
}

const sourceStreamA = fstream.Reader({
    type: "Directory",
    path: sourceA
  })
  .on('end', onEnd);

const sourceStreamB = fstream.Reader({
    type: "Directory",
    path: sourceB
  })
  .on('end', onEnd);
```

在前面的代码中，我们创建了从两个源目录（`sourceStreamA` 和 `sourceStreamB`）中读取的 `Stream` 那么对于每个源 `Stream`，我们附加一个 `end` 事件订阅者，只有当这两个目录被完全读取时，才会触发 `pack` 的 `end` 事件。

最后，合并两个 `Stream`：

```
sourceStreamA.pipe(pack, {end: false});
sourceStreamB.pipe(pack, {end: false});
```

我们将两个源文件都压缩到 `pack` 这个 `Stream` 中，并通过设定 `pipe()` 的 `option` 参数为 `{end：false}` 配置终点 `Stream` 的自动触发 `end` 事件。

这样，我们已经完成了我们简单的 `TAR` 程序。我们可以通过提供目标文件作为第一个命令行参数，然后是两个源目录来尝试运行这个实用程序：

```
node mergeTar dest.tar /path/to/sourceA /path/to/sourceB
```

在 `npm` 中我们可以找到一些可以简化 `Stream` 的合并的模块：

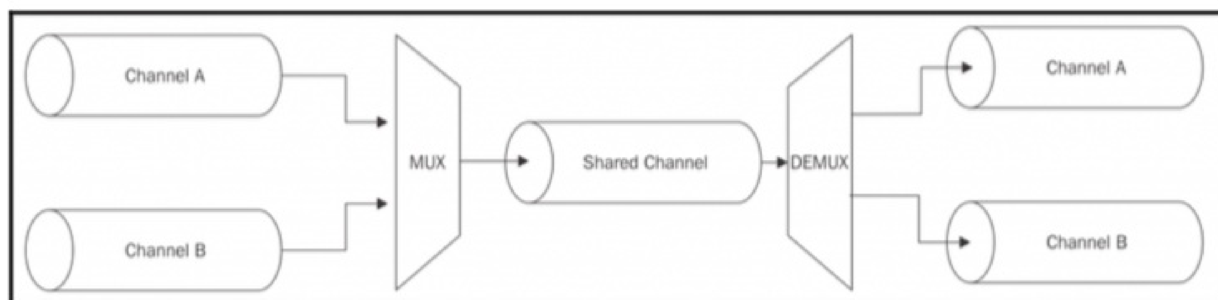
- merge-stream
- multistream-merge

要注意，流入目标 `Stream` 的数据是随机混合的，这是一个在某些类型的对象流中可以接受的属性（正如我们在上一个例子中看到的那样），但是在处理二进制 `Stream` 时通常是一个不希望这样。

然而，我们可以通过一种模式按顺序合并 `Stream`；它包含一个接一个地合并源 `Stream` ，当前一个结束时，开始发送第二段数据块（就像连接所有源 `Stream` 的输出一样）。在 `npm` 上，我们可以找到一些也处理这种情况的软件包。其中之一是multistream。

多路复用和多路分解

合并 `Stream` 模式有一个特殊的模式，我们并不是真的只想将多个 `Stream` 合并在一起，而是使用一个共享通道来传送一组数据 `Stream` 。与之前的不一样，因为源数据 `Stream` 在共享通道内保持逻辑分离，这使得一旦数据到达共享通道的另一端，我们就可以再次分离数据 `Stream` 。如图所示：



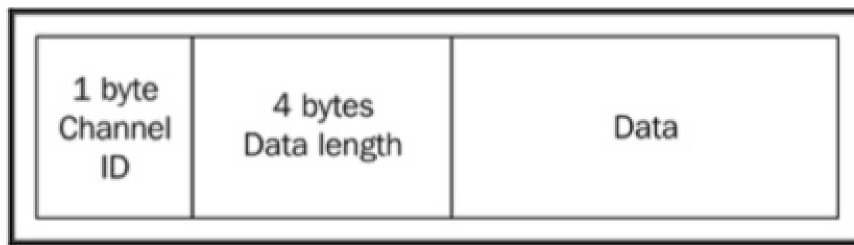
将多个 `Stream` 组合在单个 `Stream` 上传输的操作被称为多路复用，而相反的操作（即，从共享 `Stream` 接收数据重构原始的 `Stream` ）则被称为多路分用。执行这些操作的设备分别称为多路复用器和多路分解器（。 这是一个在计算机科学和电信领域广泛研究的话题，因为它是几乎任何类型的通信媒体，如电话，广播，电视，当然还有互联网本身的基础之一。 对于本书的范围，我们不会过多解释，因为这是一个很大的话题。

我们想在本节中演示的是，如何使用共享的 `Node.js Streams` 来传送多个逻辑上分离的 `Stream` ，然后在共享 `Stream` 的另一端再次分离，即实现一次多路复用和多路分解。

创建一个远程**logger**日志记录

举例说明，我们希望有一个小程序来启动子进程，并将其标准输出和标准错误都重定向到远程服务器，服务器接受它们然后保存为两个单独的文件。因此，在这种情况下，共享介质是 `TCP` 连接，而要复用的两个通道是子进程的 `stdout` 和 `stderr` 。 我们将利用分组交换的技术，这种技术与 `IP` ， `TCP` 或 `UDP` 等协议所使用的技术相同，包括将数据封装在数据包中，允许我们指定各种源信息，这对多路复用，路由，控制 流程，检查损坏的数据都十分有帮助。

如图所示，这个例子的协议大概是这样，数据被封装成具有以下结构的数据包：



在客户端实现多路复用

先说客户端，创建一个名为 `client.js` 的模块，这是我们这个应用程序的一部分，它负责启动一个子进程并实现 `Stream` 多路复用。

开始定义模块，首先加载依赖：

```
const child_process = require('child_process');
const net = require('net');
```

然后开始实现多路复用的函数：

```
function multiplexChannels(sources, destination) {
  let totalChannels = sources.length;

  for(let i = 0; i < sources.length; i++) {
    sources[i]
      .on('readable', function() { // [1]
        let chunk;
        while ((chunk = this.read()) !== null) {
          const outBuff = new Buffer(1 + 4 + chunk.length); // [
2]
          outBuff.writeUInt8(i, 0);
          outBuff.writeUInt32BE(chunk.length, 1);
          chunk.copy(outBuff, 5);
          console.log('Sending packet to channel: ' + i);
          destination.write(outBuff); // [3]
        }
      })
      .on('end', () => { //[4]
        if (--totalChannels === 0) {
          destination.end();
        }
      });
  }
}
```

`multiplexChannels()` 函数接受要复用的源 `Stream` 作为输入 和复用接口作为参数，然后执行以下步骤：

1. 对于每个源 `Stream` ，它会注册一个 `readable` 事件侦听器，我们使用 `non-flowing` 模式从流中读取数据。
2. 每读取一个数据块，我们将其封装到一个首部中，首部的顺序为： `channel ID` 为1字节（ `UInt8` ），数据包大小为4字节（ `UInt32BE` ），然后为实际数据。
3. 数据包准备好后，我们将其写入目标 `Stream` 。
4. 我们为 `end` 事件注册一个监听器，以便当所有源 `Stream` 结束时， `end` 事件触发，通知目标 `Stream` 触发 `end` 事件。

> 注意，我们的协议最多能够复用多达256个不同的源流，因为我们只有1个字节来标识 `channel` 。

```
const socket = net.connect(3000, () => { // [1]
  const child = child_process.fork( // [2]
    process.argv[2],
    process.argv.slice(3), {
      silent: true
    }
  );
  multiplexChannels([child.stdout, child.stderr], socket); // [3]

});
```

在最后，我们执行以下操作：

1. 我们创建一个新的 `TCP` 客户端连接到地址 `localhost:3000` 。
2. 我们通过使用第一个命令行参数作为路径来启动子进程，同时我们提供剩余的 `process.argv` 数组作为子进程的参数。我们指定选项 `{silent：true}` ，以便子进程不会继承父级的 `stdout` 和 `stderr` 。
3. 我们使用 `mutiplexChannels()` 函数将 `stdout` 和 `stderr` 多路复用到 `socket` 里。

在服务端实现多路分解

现在来看服务端，创建 `server.js` 模块，在这里我们将来自远程连接的 `Stream` 多路分解，并将它们传送到两个不同的文件中。

首先创建一个名为 `demultiplexChannel()` 的函数：

```
function demultiplexChannel(source, destinations) {
  let currentChannel = null;
  let currentLength = null;
  source
    .on('readable', () => { //[1]
      let chunk;
      if(currentChannel === null) {          //[2]
        chunk = source.read(1);
        currentChannel = chunk && chunk.readUInt8(0);
      }

      if(currentLength === null) {          //[3]
        chunk = source.read(4);
        currentLength = chunk && chunk.readUInt32BE(0);
        if(currentLength === null) {
          return;
        }
      }

      chunk = source.read(currentLength);        //[4]
      if(chunk === null) {
        return;
      }

      console.log('Received packet from: ' + currentChannel);

      destinations[currentChannel].write(chunk);      //[5]
      currentChannel = null;
      currentLength = null;
    })
    .on('end', () => {            //[6]
      destinations.forEach(destination => destination.end());
      console.log('Source channel closed');
    })
  ;
}
```

上面的代码可能看起来很复杂，仔细阅读并非如此；由于 `Node.js` 可读的 `Stream` 的拉动特性，我们可以很容易地实现我们的小协议的多路分解，如下所示：

1. 我们开始使用 `non-flowing` 模式从流中读取数据。
2. 首先，如果我们还没有读取 `channel ID` ，我们尝试从流中读取1个字节，然后将其转换为数字。
3. 下一步是读取首部的长度。我们需要读取4个字节，所以有可能在内部 `Buffer` 还没有足够的数据，这将导致 `this.read()` 调用返回 `null` 。在这种情况下，我们只是中断解析，然后重试下一个 `readable` 事件。
4. 当我们最终还可以读取数据大小时，我们知道从内部 `Buffer` 中拉出多少数据，所以我们尝试读取所有数据。
5. 当我们读取所有的数据时，我们可以把它写到正确的目标通道，一定要记得重

置 `currentChannel` 和 `currentLength` 变量（这些变量将被用来解析下一个数据包）。
6. 最后，当源 `channel` 结束时，一定不要忘记调用目标 `Stream` 的 `end()` 方法。

既然我们可以多路分解源 `Stream` ，进行如下调用：

```
net.createServer(socket => {
  const stdoutStream = fs.createWriteStream('stdout.log');
  const stderrStream = fs.createWriteStream('stderr.log');
  demultiplexChannel(socket, [stdoutStream, stderrStream]);
})
  .listen(3000, () => console.log('Server started'))
;
```

在上面的代码中，我们首先在 `3000` 端口上启动一个 `TCP` 服务器，然后对于我们接收到的每个连接，我们将创建两个可写入的 `Stream` ，指向两个不同的文件，一个用于标准输出，另一个用于标准错误; 这些是我们的目标 `channel` 。最后，我们使用 `demultiplexChannel()` 将套接字流解复用为 `stdoutStream` 和 `stderrStream` 。

运行多路复用和多路分解应用程序

现在，我们准备尝试运行我们的新的多路复用/多路分解应用程序，但首先让我们创建一个小的 `Node.js` 程序来产生一些示例输出; 我们把它叫做 `generateData.js` ：

```
console.log("out1");
console.log("out2");
console.error("err1");
console.log("out3");
console.error("err2");
```

首先，让我们开始运行服务端：

```
node server
```

然后运行客户端，需要提供作为子进程的文件参数：

```
node client generateData.js
```

```
node (node)
➜ node_code git:(master) ✗ node server.js
Server started
Received packet from: 0
Received packet from: 1
Source channel closed
Received packet from: 0
Received packet from: 0
Received packet from: 1
Source channel closed
```

```
..r05/node_code (zsh)
^C
➜ node_code git:(master) ✗ node client.js generateData.js
Sending packet to channel: 0
Sending packet to channel: 1
➜ node_code git:(master) ✗ ls
README.txt      client.js       generateData.js server.js       stderr.log      stdout.log
➜ node_code git:(master) ✗ cat stderr.log
err1
err2
➜ node_code git:(master) ✗ cat stdout.log
out1
out2
out3
➜ node_code git:(master) ✗ node client.js generateData.js
Sending packet to channel: 0
Sending packet to channel: 0
Sending packet to channel: 1
➜ node_code git:(master) ✗ ls
README.txt      client.js       generateData.js server.js       stderr.log      stdout.log
➜ node_code git:(master) ✗ cat stdout.log
out1
out2
out3
➜ node_code git:(master) ✗ cat stderr.log
err1
err2
➜ node_code git:(master) ✗
```

客户端几乎立马运行，但是进程结束时， `generateData` 应用程序的标准输入和标准输出经过一个 `TCP` 连接，然后在服务器端，被多路分解成两个文件。

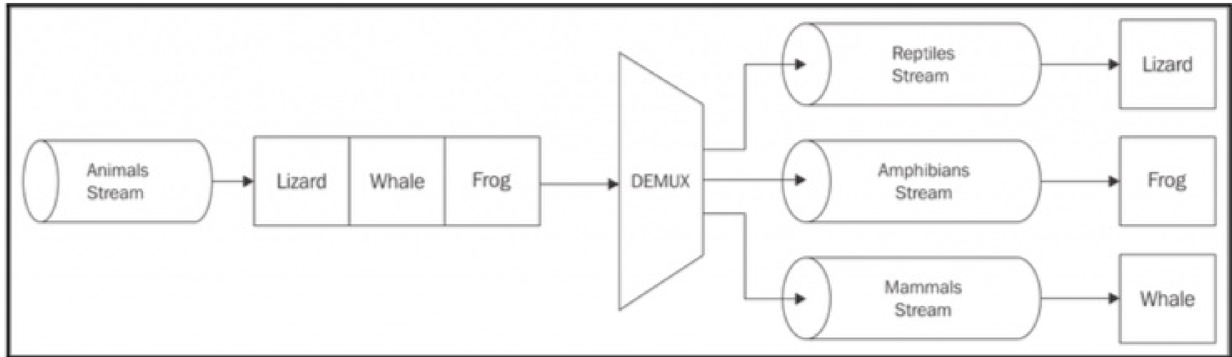
> 注意，当我们使用 `child_process.fork()` 时，我们的客户端能够启动别的 `Node.js` 模块。

对象Streams的多路复用和多路分解

我们刚刚展示的例子演示了如何复用和解复用二进制/文本 `Stream` ，但值得一提的是，相同的规则也适用于对象 `Stream` 。最大的区别是，使用对象，我们已经有了使用原子消息（对象）传输数据的方法，所以多路复用就像设置一个属

性 `channel ID` 到每个对象一样简单，而多路分解只需要读 `.channel ID` 属性，并将每个对象路由到正确的目标 `Stream` 。

还有一种模式是取一个对象上的几个属性并分发到多个目的 `Stream` 的模式 通过这种模式，我们可以实现复杂的流程，如下图所示：



如上图所示，取一个对象 `Stream` 表示 `animals` ，然后根据动物类型： `reptiles` ， `amphibians` 和 `mammals` ，然后分发到正确的目标 `Stream` 中。

总结

在本章中，我们已经对 `Node.js Streams` 及其使用案例进行了阐述，但同时也应该为编程范式打开一扇大门，几乎具有无限的可能性。我们了解了为什么 `Stream` 被 `Node.js` 社区赞誉，并且我们掌握了它们的基本功能，使我们能够利用它做更多有趣的事情。我们分析了一些先进的模式，并开始了解如何将不同配置的 `Streams` 连接在一起，掌握这些特性，从而使流如此多才多艺，功能强大。

如果我们遇到不能用一个 `Stream` 来实现的功能，我们可以通过将其他 `Streams` 连接在一起来实现，这是 `Node.js` 的一个很好的特性； `Streams` 在处理二进制数据，字符串和对象都十分有用，并具有鲜明的特点。

在下一章中，我们将重点介绍传统的面向对象的设计模式。尽管 `JavaScript` 在某种程度上是面向对象的语言，但在 `Node.js` 中，函数式或混合方法通常是首选。在阅读下一章便揭晓答案。

Design Patterns

设计模式是重复出现的问题的可重用解决方案；该术语的定义非常广泛，可以涵盖应用程序的多个领域。然而，这个术语通常与著名的面向对象模式相关联，又被称作可复用的面向对象基础方法。我们经常会将这些特定的模式集合称为传统设计模式或 `GoF` 设计模式。

在 `JavaScript` 中应用面向对象的设计模式并不像传统的面向对象的语言那样线性和形式化。我们知道， `JavaScript` 是范式化的，面向对象的，基于原型的，并且是动态类型语言；它将函数视为一等公民，并允许函数式的编程风格。这些特性使得 `JavaScript` 成为一种非常通用的语言，它为开发人员提供了巨大的力量，但同时也造成其编程风格与传统语言不同。人们总结 `JavaScript` 的编程范式，最后总结出 `JavaScript` 生态系统的模式。有很多方法可以使用 `JavaScript` 实现相同的结果。对于 `JavaScript` 的问题，解决一个问题的模式是多样化的。这种现象的一个明显的例子就是 `JavaScript` 生态系统中有丰富的框架和类库；可能没有其他语言见过这么多，尤其是现在 `Node.js` 已经给 `JavaScript` 带来了惊人的新的可能性，并创造了许多新的场景。

在这种背景下，传统的设计模式也受到 `JavaScript` 本质的影响。实现它们的方式有很多，所以它们传统的，强烈的面向对象的实现意味着它们不再是模式。在某些情况下，它们甚至是不需要的，因为我们知道， `JavaScript` 没有真正的类或抽象接口。不变的是每个模式的基本原理，解决的问题以及解决方案核心的概念。

本章探讨的设计模式如下：

- 工厂模式（ `Factory` ）
- 揭示构造模式（ `Revealing constructor` ）
- 代理模式（ `Proxy` ）
- 装饰者模式（ `Decorator` ）
- 适配器模式（ `Adapter` ）
- 策略模式（ `Strategy` ）
- 状态模式（ `State` ）
- 模板模式（ `Template` ）
- 中间件模式（ `Middleware` ）
- 命令模式（ `Command` ）

> 本章假定读者对 `JavaScript` 中继承的工作原理有一些概念。 另外请注意，在本章中，我们经常使用一般的和更直观的图来描述一个模式来代替标准的 `UML` ，因为许多模式可以有一个不仅基于类而且基于对象甚至函数的实现。

工厂模式（ `Factory` ）

我们从 `Node.js` 中最简单，最常见的设计模式工厂模式开始。

用于创建对象的通用接口

我们已经强调了这样的事实：在 `JavaScript` 中，因为函数的简单性，易用性和可拓展性，函数实例通常比纯粹的面向对象设计更受欢迎。创建新的对象实例时尤其如此。 实际上，调用一个工厂，而不是直接使用 `new` 运算符或 `Object.create()` 从一个原型创建一个新的对象，在很多方面是非常方便和灵活的。

首先，工厂允许我们将对象创建与实现分离开来；从本质上讲，一个工厂包装了一个新实例的创建，给了我们更多的灵活性和控制。 在工厂内部，我们可以使用闭包，使用原型和 `new` 运算符，使用 `Object.create()` 创建新实例，甚至根据特定条件返回不同的实例。对于对象的使用者而言，其完全不知道这个实例是怎么进行创建的。事实是，通过使用 `new` ，我们将我们的代码绑定到创建对象的一种特定方式，而在 `JavaScript` 中，可以更灵活且自由地创建对象。作为下面这个简单的例子，我们来考虑通过工厂模式创建一个 `Image` 对象：

```
function createImage(name) {
  return new Image(name);
}
const image = createImage('photo.jpeg');
```

`createImage()` 工厂可能看起来完全没有必要。为什么不直接使用 `new` 运算符来实例化 `Image` 类？像下面这行代码：

```
const image = new Image(name);
```

正如我们已经提到的，使用 `new` 将我们的代码绑定到一个特定类型的对象；对于前面的例子，绑定到 `Image` 类型的对象。工厂模式创建对象更为灵活；想象一下，如果我们想要重构 `Image` 类，把它分成更小的类，使得其支持各种图像格式。如果我们将工厂作为创建新图像的唯一方法，我们可以像如下拓展代码，而不会破坏任何现有的代码：

```
function createImage(name) {
  if (name.match(/\.jpeg$/)) {
    return new JpegImage(name);
  } else if (name.match(/\.gif$/)) {
    return new GifImage(name);
  } else if (name.match(/\.png$/)) {
    return new PngImage(name);
  } else {
    throw new Exception('Unsupported format');
  }
}
```

工厂还允许我们不暴露它创建的对象的构造函数，并防止它们被扩展或修改。在 `Node.js` 中，这可以通过仅导出工厂来实现，同时保持每个构造函数都是私有的。

强制封装机制

由于闭包，工厂也可以用来实现封装。

正如我们所知，在 `JavaScript` 中，我们没有权限修饰符（例如，我们不能声明私有变量），所以强制封装的唯一方法是通过函数作用域和闭包。 工厂可以用来实现封装，直接声明私有变量；以下面的代码为例：

```
function createPerson(name) {
  const privateProperties = {};
  const person = {
    setName: name => {
      if (!name) throw new Error('A person must have a name');
      privateProperties.name = name;
    },
    getName: () => {
      return privateProperties.name;
    }
  };
  person.setName(name);
  return person;
}
```

在前面的代码中，我们利用闭包来创建两个对象：一个表示工厂返回的公共接口的 `person` 对象，一个从外部不可访问的 `privateProperties` ，只能通过 `person` 提供的接口来操作目的。例如，在前面的代码中，要确保 `person` 的 `name` 永远不为空；如 `name` 只是 `person` 对象的属性，则不可能做到强制封装。

工厂只是我们创建私有成员变量的技术之一，事实上，也有很多其它的方法定义私有成员变量：

- 在构造函数中定义私有变量
- 使用约定，用下划线 `_` 或美元符号 `$` （但这在技术上不会阻止从外部访问成员）的属性名称前缀
- 使用 `ES2015 WeakMaps`

构建一个简单的profiler

现在，我们来看一个使用工厂模式的完整示例。让我们构建一个简单的 `profiler` ，看一个具有以下属性的对象：

- `start()` 方法，触发一个会话开始
- `end()` 方法，终止会话并记录它的执行时间，打印到控制台

我们首先创建一个名为 `profiler.js` 的文件，它将包含以下内容：

```
class Profiler {
  constructor(label) {
    this.label = label;
    this.lastTime = null;
  }
  start() {
    this.lastTime = process.hrtime();
  }
  end() {
    const diff = process.hrtime(this.lastTime);
    console.log(
      `Timer "${this.label}" took ${diff[0]} seconds and ${diff[
1]}
           nanoseconds.`
    );
  }
}
```

前面的类没有什么特别之处。我们只需使用默认的定时器来保存当 `start()` 被调用时的时间，然后计算到执行 `end()` 时的所经过的时间，并将结果打印到控制台。

现在，如果我们要在真实世界的应用程序中使用这样一个 `profiler` 来计算不同程序的执行时间，我们可以很容易想象我们将会在标准输出中产生大量的日志记录，特别是在生产环境中。我们可能想要做的是将分析信息重定向到另一个源（例如数据库），或者，如果应用程序正在生产环境下运行，则将 `profiler` 完全禁用。很明显，如果我们直接使用 `new` 运算符实例化一个 `Profiler` 对象，那么我们需要在客户端代码或 `Profiler` 对象本身中添加一些额外的逻辑，以便在不同的逻辑之间切换。我们可以使用工模式厂来抽象创建 `Profiler` 对象，这样，根据应用程序是以生产模式还是开发模式运行，我们可以返回完全正常工作的 `Profiler` 对象，或者具有相同接口的模拟对象，但方法是空函数。让我们在 `profiler.js` 模块中执行此操作，而不是导出 `Profiler` 构造函数，而只导出一个函数，即我们的工厂。以下是其代码：

```
module.exports = function(label) {
  if (process.env.NODE_ENV === 'development') {
    return new Profiler(label); // [1]
  } else if (process.env.NODE_ENV === 'production') {
    return { // [2]
      start: function() {},
      end: function() {}
    }
  } else {
    throw new Error('Must set NODE_ENV');
  }
};
```

我们创建的工厂从其中抽象了 `Profiler` 对象的创建过程：

- 如果应用程序正在开发模式下运行，我们会完全返回一个新的具有完整功能的 `Profiler` 对象。
- 如果应用程序正在生产模式下运行，则返回一个模拟对象，它的 `start()` 和 `stop()` 方法是空函数。

值得一提的是，由于 `JavaScript` 的动态输入，我们能够在一种情况下返回一个使用 `new` 运算符实例化的对象，而在另一种情况下返回一个简单的对象字面值。工厂模式可以很好地实现这一点，我们可以在工厂函数中以任何方式创建对象，可以执行额外的初始化步骤或者根据特定的条件返回不同类型的对象，而这些细节对于对象的使用者来说都是透明的。我们可以很容易地理解这种简单模式的强大。

现在我们可以使用我们的 `profiler` ，来看以下代码：

```
const profiler = require('./profiler');

function getRandomArray(len) {
  const p = profiler('Generating a ' + len + ' items long array'
);
  p.start();
  const arr = [];
  for (let i = 0; i < len; i++) {
    arr.push(Math.random());
  }
  p.end();
}
getRandomArray(1e6);
console.log('Done');
```

变量 `p` 包含我们的 `profiler` 对象实例，但是我们不知道它是 如何创建的，和在这个代码点它是如何实现的。如果我们将上面的代 码包含在 `profilerTest.js` 中，我们可以很容易地测验证这些假设。测试启用代码分析功能的程序，运行以下命令：

```
export NODE_ENV=development; node profilerTest
```

前面的命令启用开发环境的 `profiler` 然后打印分析信息到控制台。 如果我们想要看看生产环境下的 `profiler` ，我们可以运行下面的命令：

```
export NODE_ENV=production; node profilerTest
```

我们刚才展示的示例只是工厂模式的简单应用程序，但它清楚地显示了将对象的创建与实现分离的优点。

可组合的工厂函数

现在我们对如何在 `Node.js` 中实现工厂函数有了一个很好的想法，我们准备引入一个最近在 `JavaScript` 社区中引起了关注的高级模式。我们正在谈论可组合的工厂函数，它代表了一种特定类型的工厂函数，可以“组合”在一起构建新的更强大的工厂函数。它们允许我们构建继承关系较为复杂的对象十分有用。

我们可以用一个简单而有效的例子来阐明这个概念。假设我们要构建一个游戏，其中屏幕上的角色可以有许多不同的行为：可以在屏幕上移动;他们可以砍杀和射击。是的，要成为一个角色，他们应该有一些基本的属性，如生命值，屏幕上的位置和角色类型。

我们要定义几种类型的角色，每一种特定的行为：

- `Character` ：具有生命值，位置和名字的基础角色
- `Mover` ：可移动的角色
- `Slasher` ：可砍杀他人的角色
- `Shooter` ：能够射击的角色（只要有子弹就可以成为 `Shooter` ！）

理想情况下，我们可以定义新的角色类型，结合现有角色的不同行为。 我们希望有绝对的自由，例如，我们希望在现有的基础上定义这些新的类型：

- `Runner` ：可移动的角色
- `Samurai` ：可移动和砍杀他人的角色
- `Sniper` ：不能移动但能射击的角色
- `Gunslinger` ：可以移动和射击的角色
- `Western Samurai` ：可移动、砍杀他人和射击的角色

正如你所看到的，我们希望完全自由地结合每个基本类型的特征，所以现在应该很明显的是我们不能用类和继承来简单地模拟这个问题。

相反，我们将使用可组合的工厂函数，特别是我们可以使用stamp模块。

这个模块提供了一个直观的接口来定义工厂函数，可以组合起来构建新的工厂函数。基本上，它允许我们定义工厂函数，通过使用方便流畅的接口来描述它们，这些工厂函数将生成具有一组特定属性和方法的对象。

让我们看看如何通过 `stamp` 定义我们的游戏的基本角色。我们将从基础的角色开始：

```
const stampit = require('stampit');
const character = stampit().
props({
  name: 'anonymous',
  lifePoints: 100,
  x: 0,
  y: 0
});
```

在前面的代码片段中，我们定义了角色的工厂函数，它可以用来创建基本角色的新实例。每个角色将具有以下属性：name，lifePoints，x和y，默认值分别为 `'anonymous'` ， `100` ， `0` 和 `0` 。使用 `stampit` 的 `props` 方法可以定义这些属性。 要使用这个工厂函数，我们可以这样做：

```
const c = character();
c.name = 'John';
c.lifePoints = 10;
console.log(c); // { name: 'John', lifePoints: 10, x:0, y:0 }
```

现在，让我们来定义 `mover` 工厂函数：

```
const mover = stampit()
  .methods({
    move(xIncr, yIncr) {
      this.x += xIncr;
      this.y += yIncr;
      console.log(`${this.name} moved to [${this.x}, ${this.y}]`
);
    }
  });
```

在这种情况下，我们使用 `stampit` 的 `methods` 函数来声明这个工厂函数产生的对象中所有可用的方法。 对于我们的 `Mover` 定义，我们有一个 `move` 函数可以增加实例的 `x` 和 `y` 的位置。 请注意，我们可以从方法内使用关键字 `this` 来访问实例属性。

现在我们已经理解了基本的概念，我们可以很容易地添加 `slasher` 和 `shooter` 类型的工厂函数定义：

```
const slasher = stampit()
  .methods({
    slash(direction) {
      console.log(`${this.name} slashed to the ${direction}`);
    }
  });
const shooter = stampit()
  .props({
    bullets: 6
  })
  .methods({
    shoot(direction) {
      if (this.bullets > 0) {
        --this.bullets;
        console.log(`${this.name} shoot to the ${direction}`);
      }
    }
  });
```

注意到我们如何使用 `props` 和 `methods` 来定义我们的 `shooter` 工厂函数。

现在我们已经定义了所有的基本类型，我们准备将它们组合起来创建新的更为复杂的工厂函数。

```
const runner = stampit.compose(character, mover);
const samurai = stampit.compose(character, mover, slasher);
const sniper = stampit.compose(character, shooter);
const gunslinger = stampit.compose(character, mover, shooter);
const westernSamurai = stampit.compose(gunslinger, samurai);
```

`stampit.compose()` 方法定义了一个新的组合的工厂函数，它的作用是根据组合工厂函数的方法和属性生成一个对象。 正如你所看到的那样，这是一个强大的机制，使我们能够自由地创建和组合工厂函数。

接下来我们实例化一个新的 `westernSamurai` 。

```
const gojiro = westernSamurai();
gojiro.name = 'Gojiro Kiryu';
gojiro.move(1, 0);
gojiro.slash('left');
gojiro.shoot('right');
```

这将产生以下输出：

```
Yojimbo moved to [1, 0]
Yojimbo slashed to the left
Yojimbo shoot to the right
```

实际应用场景

正如我们所说的，工厂模式在 `Node.js` 中非常流行，许多软件包只提供用于创建新实例的工厂；常见一些例子如下：

- Dnode： `Node.js` 的远程程序调用（ `RPC` ）库。如果我们查看它的源代码，我们会看到它的逻辑实际上是实现成一个名为 `D` 的类；然而，实例并没有暴露给外界，因为唯一的接口是工厂，这使我们能够使用它创建类的新实例。你可以看看它的源代码。

- Restify：这是一个构建 `REST API` 的框架，它允许我们使用 `restify.createServer()` 工厂函数创建一个服务器的新实例，该工厂在内部创建一个新的实例 `Server` 类（不导出）。你可以看看它的源代码。

其他模块公开了一个类和一个工厂，但将工厂作为创建新实例的主要方法或最方便的方法；一些例子如下：

- http-proxy：这是一个可编程 `HTTP` 的代理库，用 `httpProxy.createProxyServer(options)` 创建新的实例。
- `Node.js` 核心模块之 `HTTP` ：这是新实例主要使用 `http.createServer()` 创建的地方，但这实际上是 `new http.Server()` 的简写方式。
- bunyan：这是一个广泛使用的日志记录库；在其 `README` 文件中，要求这个仓库的 `contributors` 需要使用工厂函数 `bunyan.createLogger()` 作为创建新实例的主要方法，即使这相当于运行 `new bunyan()` 。

其他一些模块也提供了一个工厂函数来封装其组件实例的创建。常见的例子是 `through2` 和 `from2` （我们在 `Chapter 5-Coding with Streams` 看到过它），它允许我们使用工厂方法简化新 `Streams` 的创建，从而显式地使用继承和 `new` 运算符。

还有一些使用 `stamp` 规范和组合工厂模式的模块，可以看看react-stampit，它在前端使用组合工厂模式，使您可以轻松地组合组件功能，remitter，一个基于 `Redis` 的 `pub / sub` 模块。

揭示构造函数模式（ `Revealing constructor` ）

揭示构造函数模式是一个相对较新的模式，在 `Node.js` 社区和 `JavaScript` 中越来越受到重视，特别是因为它在一些核心库（如 `Promise` ）中使用。

我们已经在 `Chapter4-Asynchronous Control Flow Patterns with ES2015 and Beyond` 中隐含地看到了这种模式，但是我们再回过头来分析一下 `Promise` 构造函数，以更详细地描述它：

```
const promise = new Promise(function(resolve, reject) {
  // ...
});
```

正如你所看到的， `Promise` 接受一个函数作为构造函数的参数，这被称为执行函数。这个函数是由 `Promise` 构造函数的内部实现调用的，它提供给构造函数，用于处理 `pending` 状态的 `promise` 的内部状态。换句话说，它确定了一个方式来调用 `resolve` 和 `reject` 函数， `promise` 遵循这个机制，调用 `resolve` 和 `reject` 来改变对象的内部状态。

这样做的好处是只有构造函数的参数函数才有权 `resolve` 和 `reject` ，一旦构造了 `Promise` 对象，就可以安全地传递；没有其他代码将能够调用 `resolve` 或 `ject` ，来改变 `Promise` 的内部状态。 这就是为什么这个模式被 `Domenic Denicola` 的一篇博客文章命名为揭示构造函数模式的原因。

一个只读的 `event emitter`

在这一段中，我们将使用揭示构造函数模式来构建一个只读的 `event emitter` ，这是一种特殊类型的 `event emitter` ，在这个 `event emitter` 内部方法，不允许调用 `emit` 方法，只有传递给构造函数的函数参数才能够调用 `emit` 方法。

让我们将 `Roee` 类的代码写入名为 `roee.js` 的文件中：

```
const EventEmitter = require('events');
module.exports = class Roee extends EventEmitter {
  constructor(executor) {
    super();
    const emit = this.emit.bind(this);
    this.emit = undefined;
    executor(emit);
  }
};
```

在这个简单的类中，我们扩展了核心模块 `EventEmitter` 类，其接受一个 `executor` 函数作为构造函数的唯一参数。

在构造函数内部，我们调用 `super` 函数来确保通过调用其父构造函数来正确地初始化 `event emitter` ，然后保存 `emit` 函数的备份，并通过为其分配 `undefined` 来删除它。

最后，我们通过传递 `emit` 方法备份作为参数来调用 `executor` 函数。

这里要了解的重要一点是，在 `undefined` 被分配给 `emit` 方法之后，我们不能再从代码的其他部分调用它了。 我们的 `emit` 的备份版本被定义为一个局部变量，只会被转发给执行器函数。这个机制使我们能够仅在 `executor` 函数内使用 `emit` 。

现在让我们使用这个新类来创建一个简单的 `ticker` ，一个每秒发出一个 `tick` 并记录所有 `tick` 发出的数量的类。

这将是我们新的 `ticker.js` 模块的内容：

```
const Roee = require('./roee');
const ticker = new Roee((emit) => {
  let tickCount = 0;
  setInterval(() => emit('tick', tickCount++), 1000);
});
module.exports = ticker;
```

正如你在这里看到的，代码量并不大。 我们实例化一个新的 `Roee` ，并在 `executor` 函数内传递 `emit` 作为参数。正是因为我们的 `executor` 函数接收 `emit` 作为参数，所以我们可以使用它每秒发出一个新的 `tick` 事件。

现在我们举例说明如何使用这个模块：

```
const ticker = require('./ticker');
ticker.on('tick', (tickCount) => console.log(tickCount, 'TICK'))
;
// ticker.emit('something', {}); <-- This will fail
```

我们使用与任何其他基于 `event emitter` 的对象相同的 `ticker` 对象，我们可以用 `on` 方法附加任意数量的监听器，但是在这种情况下，如果我们尝试使用 `emit` 方法，那么我们的代码将抛出异常 `TypeError: ticker.emit is not a function` 。

> 即使这个例子在展示如何使用揭示构造函数模式，但值得一提的是这个事件发生器的只读功能并不是完美的，并且仍然有可能以几种方式绕过它。例如，我们仍然可以通过直接使用原型上的 `emit` 在我们的 `ticker` 实例上发出事件，如下所示：

```
require('events').prototype.emit.call(ticker, 'someEvent', {});
```

实际应用场景

即使这种模式非常有趣和智能，但实际上，除了 `Promise` 构造函数以外，很难找到常见的应用实例。

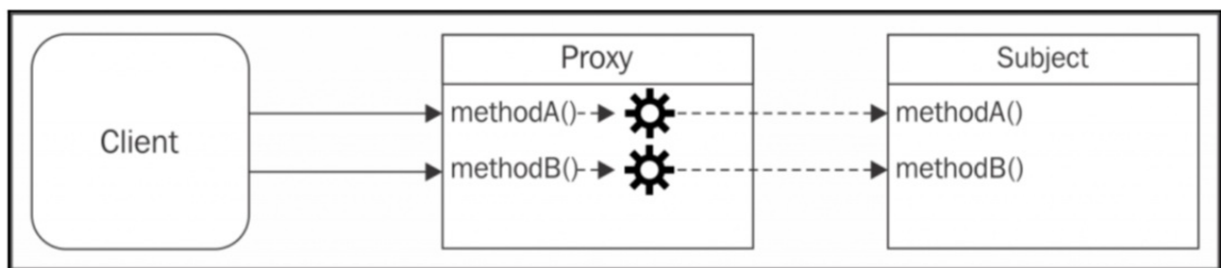
值得一提的是，现在 `Streams` 议案中有一个新的规范，可以尝试使用揭示构造函数模式替代现今的模板模式，以便能够描述各种 `Streams` 对象的行为：可以看 https://streams.spec.whatwg.org/

另外需要指出的是，在之前 `Chapter 5-Coding with Streams` 当我们实现了 `ParallelStream` 类的时候。这个类作为构造函数参数接受 `userTransform` 函数作为参数（ `executor` ）。

即使在这种情况下， `executor` 函数在构建时不被调用，但在 `Streams` 的内部 `_transform()` 方法中，揭示构造函数模式的一般概念仍然有效。实际上，这种方法允许我们在创建一个新的 `ParallelStream` 实例时，将 `Streams` 的一些内部方法（例如 `push` 函数）暴露给 `executor` 函数，使得我们在调用构造函数创建 `ParallelStream` 实例时执行与内部方法相关的一些操作。

代理模式（ `Proxy` ）

代理是一个控制访问另一个被称为主体对象的对象。代理对象和主体对象有一套相同的接口，这使得在使用代理的过程中，对于使用者而言是透明的。这种模式称为代理模式。代理拦截了所有要在主体对象上进行的操作，并增强或补充主体对象的行为。如图所示：



上图给我们展示了代理对象和主体对象具有相同的接口，以及这对客户端来说是如何完全透明的，客户端可以互换地使用其中一个。代理将每个操作转发给主体，通过额外的预处理或后处理来增强其行为。

> 要注意的是，我们并不是在讨论对于不同类需要实现不同的代理。代理模式要求代理对象需要保持各自主体的状态。

代理在几种情况下是有用的；例如，考虑以下几点情况：

- 数据验证：在代理向主体转发数据前验证其数据输入的合法性。
- 安全性：代理验证客户端是否有权限，仅仅当有权限时才会向主体对象发送相关请求。
- 缓存：代理对象保存内部缓存，仅仅当缓存未命中时才向主体对象发送相关请求。
- 懒加载：如果主体对象的创建需要消耗大量资源，代理可以推迟创建主体对象的时机，仅仅当需要主体对象时才创建主体对象。
- 日志：代理拦截方法和对应的参数调用，并在他们执行前后实现日志打印。
- 远程对象：代理可以接收远程对象，并使得其呈现为本地对象。

当然，代理模式还有更多的应用，但以上这些应该能让我们了解其主要用途。

实现代理的技术

当代理一个对象时，我们可以拦截所有的方法，或者只拦截其中的一些，而把其余的直接委托给主体对象。有几种方法可以实现这一点。让我们来分析其中的一些方法。

对象组合

对象组合是一种将对象与另一个对象组合起来的技术，便于扩展或使用其中一个对象功能。对于代理模式而言，创建具有与主体对象相同接口的新对象，并且对该主体的引用以实例变量或闭包变量的形式存储在代理内部。

主体对象可以在创建时从客户端注入，也可以由代理自己创建。

以下是使用伪类和工厂模式创建代理对象的一个例子：

```
function createProxy(subject) {
  const proto = Object.getPrototypeOf(subject);

  function Proxy(subject) {
    this.subject = subject;
  }
  Proxy.prototype = Object.create(proto);
  //proxied method
  Proxy.prototype.hello = function() {
    return this.subject.hello() + ' world!';
  };
  //delegated method
  Proxy.prototype.goodbye = function() {
    return this.subject.goodbye
      .apply(this.subject, arguments);
  };
  return new Proxy(subject);
}
module.exports = createProxy;
```

为了使用对象组合实现代理，我们必须拦截我们需要的方法（比如 `hello()` ），对于我们不需要的方法，则委托给主体对象调用（例如 `goodbye()` 方法）。

前面的代码也显示了主体对象有一个原型的特定情况，我们希望维护正确的原型链，以便执行代理 `instanceof Subject` 将返回 `true` ；我们使用继承来实现这一点。

这只是一个额外的步骤，当我们想要保持原型链时，才需要这个步骤，这对于改进代理的兼容性有用的。

但是，由于 `JavaScript` 具有动态类型，大多数情况下我们可以避免使用继承，并使用更直接的方法。例如，前面的代码中提供的代理的另一种实现，可能只使用对象字面量和工厂模式：

```
function createProxy(subject) {
  return {
    // 代理方法
    hello: () => (subject.hello() + ' world!'),
    // 委托方法
    goodbye: () => (subject.goodbye.apply(subject, arguments))
  };
}
```

> 如果我们想创建一个委托其大部分方法的代理，那么使用称为delegates自动生成这些代理会很方便。

对象增强

对象增强是实现代理模式最佳的方式，通过用在代理对象上实现替换方法来直接修改对象；看下面的例子：

```
function createProxy(subject) {
  const helloOrig = subject.hello;
  subject.hello = () => (helloOrig.call(this) + ' world!');
  return subject;
}
```

当我们需要实现的代理只有一个或几个方法的时候，这个技术绝对是最方便的，但是它有一个缺点，就是直接修改主体对象。

不同技术的比较

对象组合被认为是创建代理的最安全的方式，因为它可以在不改变主体对象的原始行为的情况下创建代理。它唯一的缺点是我们必须手动委托所有的方法，即使我们只想代理其中的一个方法。如果需要的话，我们可能还必须委托访问主体对象的属性。

> 对象原型能够通过使用 `Object.defineProperty()` 被委托，可以查看Object.defineProperty()的文档

对于对象增强而言，可能我们并不是总想要修改主体对象，但是它没有出现在委派方法中出现的种种不便。为此，对象增强是 用 `JavaScript` 实现代理最实用的方式，如果修改主体对象不会导致大问题，对象增强是首选的技术。

然而，至少有一种情况下，对象组合几乎是必需的；这就是我们想要控制主体对象的初始化时，例如，只在需要它的时候才创建(懒加载)。

> 值得指出的是，通过使用工厂函数（在我们的例子中是 `createProxy()` ），我们可以将代码从用于生成代理。

创建一个可写入的日志流

为了实现一个代理模式的例子，现在我们创建一个可写入 `Streams` 的例子，通过拦截对 `write()` 函数的全部调用，然后对于每次调用记录一条信息。我们会使用对象组合来实现我们的代理，创建一个 `loggingWritable.js` 文件来实现：

```
const fs = require('fs');

function createLoggingWritable(writableOrig) {
  const proto = Object.getPrototypeOf(writableOrig);

  function LoggingWritable(writableOrig) {
    this.writableOrig = writableOrig;
  }

  LoggingWritable.prototype = Object.create(proto);

  LoggingWritable.prototype.write = function(chunk, encoding, ca
llback) {
    if(!callback && typeof encoding === 'function') {
      callback = encoding;
      encoding = undefined;
    }
    console.log('Writing ', chunk);
    return this.writableOrig.write(chunk, encoding, function() {
      console.log('Finished writing ', chunk);
      callback && callback();
    });
  };

  LoggingWritable.prototype.on = function() {
    return this.writableOrig.on
      .apply(this.writableOrig, arguments);
  };

  LoggingWritable.prototype.end = function() {
    return this.writableOrig.end
      .apply(this.writableOrig, arguments);
  };

  return new LoggingWritable(writableOrig);
}
```

在前面的代码中，我们创建了一个返回代理对象的代理版本的工厂函数，工厂函数需要传递主体对象作为参数。我们覆盖了 `write()` 方法，每次调用 `write()` 时都会将消息记录到标准输出，并且每次异步操作完成时都会记录消息。这也是创建

异步函数的代理的一个很好的例子，因为我们知道代理回调函数也是必要的。这是在诸如 `Node.js` 的平台中要考虑的重要细节。 其余的方法 `on()` 和 `end()` 只是委托给原来的可写入的 `Streams` （为了让代码更加精简，我们没有考虑可写入接口的其他方法）。 现在我们可以在 `loggingWritable.js` 模块中添加几行代码来测试我们刚创建的代理：

```
const writable = fs.createWriteStream('test.txt');
const writableProxy = createLoggingWritable(writable);

writableProxy.write('First chunk');
writableProxy.write('Second chunk');
writable.write('This is not logged');
writableProxy.end();
```

因为使用对象组合，这个代理不会改变 `Streams` 或者它的外部行为的原有接口，如果我们运行前面的代码，我们现在回看到每个数据块写入 `Streams` 的过程透明地写到控制台中。

代理生态-函数钩子和AOP

在众多的设计模式中，代理模式在 `Node.js` 以及生态系统中都是相当流行的模式。实际上，我们可以找到几个允许我们简化代理创建的库，大部分时间利用对象增强作为实现方法。在社区中，这个模式也可以称为函数挂钩，或者有时称为面向方面的编程（ `AOP` ），它实际上是代理的一个常见应用领域。在 `AOP` 中，这些库通常允许开发人员为特定方法（或一组方法）设置执行前或执行后钩子，这些方法允许我们分别在执行建议的方法之前和之后执行自定义代码。

有时，代理也被称为中间件，因为在中间件模式中（我们将在本章后面会看到），它们允许我们对函数的输入/输出进行预处理和后处理。有时，他们也允许使用类似中间件的管道为同一方法注册多个钩子。

在 `npm` 上有几个库允许我们用很少的努力实现函数钩子。其中有hooks，hooker，和meld。

ES2015的Proxy

`ES2015` 规范引入了一个名为 `Proxy` 的全局对象，它可以从开始在 `Node.js v6.0` 中使用。

`Proxy API` 包含一个 `Proxy` 构造函数，它接受一个 `target` 和一个 `handler` 作为参数：

```
const proxy = new Proxy(target, handler);
```

这里，`target` 表示应用代理的对象（我们的规范定义的主体对象），而 `handler` 是定义代理行为的特殊对象。

`handler` 对象包含一系列具有预定义名称的可选方法，这些方法称为陷阱方法（例如，`apply`，`get`，`set` 和 `has`），这些方法在代理实例上执行相应的操作时会自动调用。

为了更好地理解这个 `API` 的工作原理，我们来看一个例子：

```
const scientist = {
  name: 'nikola',
  surname: 'tesla'
};
const uppercaseScientist = new Proxy(scientist, {
  get: (target, property) => target[property].toUpperCase()
});
console.log(uppercaseScientist.name, uppercaseScientist.surname)
;
// NIKOLA TESLA
```

在这个例子中，我们使用 `Proxy API` 来拦截对目标对象 `scientist` 属性的所有访问，并将属性的原始值转换为大写字符串。

如果你仔细看看这个例子，你可能会注意到这个 `API` 的一些特别的东西：它允许我们拦截对目标对象的通用属性的访问。这是可能的，因为 `API` 不仅仅是一个简单的包装来促进代理对象的创建，就像我们在本章前面部分所定义的那样；相反，它是深入集成到 `JavaScript` 语言本身的一个特性，它使开发人员能够拦截和定制可以在对象上执行的许多操作。这个特性开创了一些新的有趣的场景，这些场景在元编程，运算符重载和对象虚拟化之前是不容易实现的。

我们来看另一个例子来阐述这个概念：

```
const evenNumbers = new Proxy([], {
  get: (target, index) => index * 2,
  has: (target, number) => number % 2 === 0
});
console.log(2 in evenNumbers); // true
console.log(5 in evenNumbers); // false
console.log(evenNumbers[7]); // 14
```

在这个例子中，我们正在创建一个包含所有偶数的虚拟数组。它可以作为常规数组使用，这意味着我们可以使用常规数组语法访问数组中的每一项（例如，`evenNumbers[7]`），或者使用 `in` 运算符检查数组中是否存在元素（例如，偶数中有 `2` 个）。该数组被认为是虚拟的，因为我们从不在其中存储数据。看一下这个实现，这个代理使用一个空的数组作为目标，然后在处理程序中定义陷阱 `get` 和 `has`：

get陷阱拦截对数组元素的访问，返回给定索引的双倍，而是拦截 `in` 运算符的用法，并检查给定的数字是否是偶数。

`Proxy API` 支持一些其他有趣的陷阱，如 `set` ， `delete` ，和 `construct` ，并允许我们创建代理，可以根据需要撤销，禁用所有的陷阱和恢复 `target` 对象的原始行为。

分析所有这些功能超出了本章的范围。这里重要的是理解 `Proxy API` 提供了一个强大的基础，以便在需要时利用代理模式。

> 如果您想了解更多关于 `Proxy API` 的知识并发现其所有功能和陷阱方法，请参阅 `Mozilla` 的本文中的更多内容：https://developer.mozilla.org/it/docs/Web/JavaScript/Reference/Global_Objects/Proxy 。 另一个很好的来源是来自 `Google` 的详细文章：https://developers.google.com/web/updates/2016/02/es2015-proxies 。

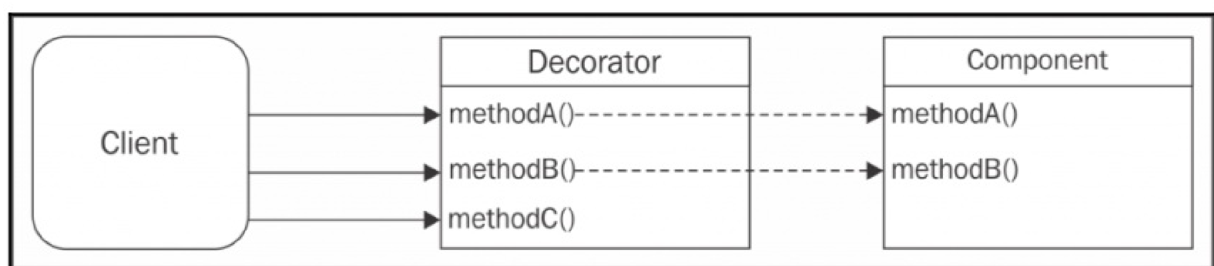
实际应用场景

Mongoose是 `MongoDB` 的一个流行的对象文档映射（ `ODM` ）库。 在内部，它使用hooks为 `init` ， `validate` ， `save` 和 `remove` 函数提供预处理和后处理的钩子函数。有关官方文档，请参阅Mongoose的官方文档。

装饰者模式（ `Decorator` ）

装饰者模式是一种结构模式，由动态增加现有对象的行为组成。 这与经典继承不同，因为行为不会添加到同一类的所有对象中，而只会添加到明确装饰的实例中。

从实现的角度来看，它与代理模式非常相似，但不是增强或修改对象的现有接口的行为，而是使用新功能增强它，如下图所示：



在上图中， `Decorator` 对象通过添加 `methodC()` 操作来扩展 `Component` 对象。

通常将现有的方法委托给装饰对象，而无需进一步处理。 当然，如果需要，我们可以轻松地组合代理模式，以便对现有方法的调用也可以被拦截和操纵。

实现装饰者模式的技巧

虽然代理模式和装饰者模式在概念上是两种不同的模式，不同的模式，他们实际上共享相同的实施策略。让我们来回顾一下。

对象组合

使用组合，被装饰的组件通常被包裹在继承它的新对象周围。在这种情况下，装饰器只需要定义新的方法，而将现有的方法委托给原始组件：

```
function decorate(component) {
  const proto = Object.getPrototypeOf(component);

  function Decorator(component) {
    this.component = component;
  }
  Decorator.prototype = Object.create(proto);
  // 新方法
  Decorator.prototype.greetings = function() {
    return 'Hi!';
  };
  // 委托方法
  Decorator.prototype.hello = function() {
    return this.component.hello.apply(this.component, arguments)
;
  };
  return new Decorator(component);
}
```

对象增强

装饰者模式也可以通过简单地将新方法直接附加到被装饰对象来实现，如下所示：

```
function decorate(component) {
  // 新方法
  component.greetings = () => {
    return component;
  };
}
```

对于使用对象组合和对象增强在代理模式的缺陷也同样适用于装饰者模式。现在让我们通过一个实例来练习装饰者模式！

装饰一个 `LevelUP` 数据库

在我们开始编码下一个例子之前，先说一下我们现在要使用的模块 `LevelUP` 。

介绍 `LevelUP` 和 `LevelDB`

LevelUP是 `Google` 的 `LevelDB` 上的一个 `Node.js` 包装器，它是最初为了在 `Chrome` 浏览器中实现 `IndexedDB` 而创建的键/值存储库，但它远不止于此。由于其极简主义和可扩展性， `LevelDB` 被 `Dominic Tarr` 定义为“Node.js的数据库”。像 `Node.js` 一样， `LevelDB` 提供了非常高效的性能，只有最基本的一组功能，允许开发人员在其上构建任何类型的数据库。 `Node.js` 社区（在这种情况下是 `Rod Vagg` ）并没有错过通过创建 `LevelUP` 将这个数据库的强大功能带入 `Node.js` 的机会。作为 `LevelDB` 的包装，它演变成支持从内存存储到其他 `NoSQL` 数据库（如 `Riak` 和 `Redis` ）到Web存储引擎（如 `IndexedDB` 和 `localStorage` ）的几种后端，使我们可以在服务器和客户端上使用相同的 `API` ，开放了一些非常有趣的场景。

现在， `LevelUP` 已经形成了一个完整的生态系统，由插件和模块组成，扩展了微型核心，实现复制，二级索引，实时更新，查询引擎等功能。而且，完整的数据库是建立在 `LevelUP` 之上的，包括 `CouchDB` 的克隆（例如PouchDB和CouchUP），甚至包括图数据库，levelgraph，它可以在 `Node.js` 和浏览器上工作！

> 了解更多关于LevelUP生态系统的信息： https://github.com/rvagg/node-levelup/wiki/Modules

实现一个 `LevelUP` 插件

在下一个示例中，我们将展示如何使用装饰者模式为 `LevelUP` 创建一个简单的插件，特别是使用对象增强技术，这是最简单但仍然最实用且最有效的方法来装饰对象能力。

> 为了方便，我们将使用level，它捆绑了 `levelup` 和名为 `leveldown` 的默认适配器，后者使用 `LevelDB` 作为后端。

我们想要构建的是一个 `LevelUP` 的插件，它允许我们在每次将具有特定模式的对象保存到数据库时接收通知。 例如，如果我们订阅 `{a: 1}` 这种类型的对象，我们希望在保存诸如 `{a: 1, b: 3}` 或 `{a: 1, c: 'x'}` 的对象时收到通知进入数据库。

我们开始通过创建一个名为 `levelSubscribe.js` 的新模块来构建我们的小插件。然后我们将插入下面的代码：

```
module.exports = function levelSubscribe(db) {
  db.subscribe = (pattern, listener) => {       //[1]
    db.on('put', (key, val) => {         //[2]
      const match = Object.keys(pattern).every(
        k => (pattern[k] === val[k])     //[3]
      );

      if(match) {
        listener(key, val);            //[4]
      }
    });
  };
  return db;
};
```

这就是我们的插件，它非常简单。让我们简单看看在前面的代码中会发生什么：

1. 用一个名为 `subscribe()` 的新方法来装饰 `db` 对象。并且使用对象增强的方式直接将方法直接附加到提供的 `db` 实例。
2. 监听对数据库进行的任何 `put` 操作。
3. 执行了一个非常简单的模式匹配算法，它验证了所提供的模式中的所有属性。
4. 一旦匹配成功，通知监听者。

现在让我们来创建一些代码 - 在一个名为 `levelSubscribeTest.js` 的新文件中 - 试用我们的新插件：

```
const level = require('level'); // [1]
const levelSubscribe = require('./levelSubscribe'); // [2]

let db = level(__dirname + '/db', {valueEncoding: 'json'});

db = levelSubscribe(db);
db.subscribe(
  {doctype: 'tweet', language: 'en'}, // [3]
  (k, val) => console.log(val)
);

db.put('1', {doctype: 'tweet', text: 'Hi', language: 'en'}); //[
4]
db.put('2', {doctype: 'company', name: 'ACME Co.'});
```

这就是我们在前面的代码中所做的：

1. 首先，我们初始化我们的 `LevelUP` 数据库，选择存储文件的目录以及这些值的默认编码。
2. 然后，我们附上我们的插件，它装饰原始的 `db` 对象。
3. 此时，我们准备使用由我们的插件提供的新特性，即 `subscribe()` 方法，在那里我们订阅包含 `doctype: "tweet"` 和 `language: "en"` 属性的对象。

4. 最后，我们使用 `put` 来保存数据库中的一些值。 第一个调用将触发与订阅相关的回调，我们将看到存储在控制台中的对象。这是因为在这种情况下，对象匹配订阅。相反，第二个调用将不会生成任何输出，因为存储的对象将不符合订阅条件。

这个例子展示了装饰者模式在其最简单实现中的实际应用：对象 增强。它看起来像一个普通的模式，但是如果使用得当，它无疑是很强大的。

> 为了简单起见，我们的插件只能与 `put` 操作结合使用，但是实际上对于batch操作也可以使用装饰者模式来进行拓展。

实际应用场景

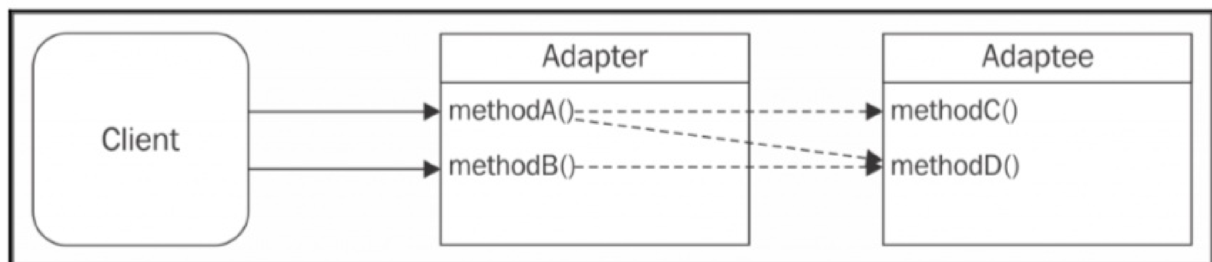
有关更多使用装饰器的更多示例，我们可能要阅读一些更多的 `LevelUP` 插件的代码：

- level-inverted-index：这是一个插件，它将倒排索引添加到 `LevelUP` 数据库中，允许我们在存储在数据库中的值上执行简单的文本搜索。
- level-plus：这是一个将原子更新添加到 `LevelUP` 数据库的插件。

适配器模式（`Adapter`）

适配器模式允许我们使用不同的接口访问对象的功能。顾名思义，它适配一个对象，以便它可以被不同接口调用。

下图阐述了适配器模式情况：



上图显示了 `Adapter` 对象的本质是 `Adaptee` 对象的包装，暴露了一个不同的接口。该图还突出显示了 `Adapter` 对象的操作也可以是 `Adaptee` 对象上一个或多个方法调用的组合。从实现的角度来看，最常见的技术是组合，其中 `Adapter` 的方法为 `Adaptee` 的方法提供了桥梁。这个模式非常简单，让我们举例说明。

通过文件系统 `API` 使用 `LevelUP`

现在我们将围绕 `LevelUP API` 构建一个适配器，将其转换为与核心 `fs` 模块兼容的接口。 特别是，我们将确保每次调用 `readFile()` 和 `writeFile()` 都将转化为对 `db.get()` 和 `db.put()` 的调用。 这样我们就可以使用一个 `LevelUP` 数据库作为简单文件系统操作的存储后端。

首先创建一个名为 `fsAdapter.js` 的新模块。 我们将首先加载依赖关系并导出我们要用来构建适配器 `createFsAdapter()` 工厂函数：

```
const path = require('path');
module.exports = function createFsAdapter(db) {
  const fs = {};
  // ...
}
```

接下来，我们会在工厂函数内实现 `readFile()` 函数，确保它的接口与 `fs` 模块某一个原有函数是兼容的：

```
fs.readFile = function(filename, options, callback) {
  if (typeof options === 'function') {
    callback = options;
    options = {};
  } else if (typeof options === 'string') {
    options = {
      encoding: options
    };
  }
  db.get(path.resolve(filename), { //[1]
      valueEncoding: options.encoding
    },
    function(err, value) {
      if (err) {
        if (err.type === 'NotFoundError') { //[2]
          err = new Error('ENOENT, open \'' + filename + '\'');
          err.code = 'ENOENT';
          err.errno = 34;
          err.path = filename;
        }
        return callback && callback(err);
      }
      callback && callback(null, value); //[3]
    }
  );
};
```

在前面的代码中，我们不得不做一些额外的工作，以确保我们的新函数的行为尽可能接近原始的 `fs.readFile()` 函数。

该函数所执行的步骤如下所述：

1. 为了从 `db` 类提取一个文件，我们使用 `filename` 作为 `key` 调用 `db.get()` ，使用它的完整的路径（使用 `path.resolve()` ）。我们设置 `valueEncoding` 的值，等于作为输入参数的任意 `encoding` 选项。
2. 如果 `key` 在数据库没有找到，我们创建一个带有 `ENOENT` 作为错误码

的 `error` ，错误码 `fs` 模块用来表示一个不存在的文件。其余的 `error` 转发给 `callback` 。
3. 如果 `key-value` 对从数据库中成功提取，我们会使用 `callback` 将 `value` 返回给调用者。

我们可以看到，我们创建的函数相当粗糙， 它并不可能成为 `fs.readFile()` 函数的完美替代品，但是在最常见的情况下它肯定会完成它的工作。

为了完成我们的 `fs` 模块适配器插件，现在让我们看看如何实现 `writeFile()` 函数：

```
fs.writeFile = (filename, contents, options, callback) => {
  if (typeof options === 'function') {
    callback = options;
    options = {};
  } else if (typeof options === 'string') {
    options = {
      encoding: options
    };
  }
  db.put(path.resolve(filename), contents, {
    valueEncoding: options.encoding
  }, callback);
}
```

另外，在这种情况下，我们没有进行完美的包装和适配，因为我们忽略了一些选项，比如对于文件权限（ `options.mode` ）来说，我们会按照原样传递从数据库收到的任何错误。

最后，我们只需要返回 `fs` 对象并使用下面这行代码关闭工厂函数：

```
return fs;
```

`fsAdapter.js` 完整代码：

```
const path = require('path');

module.exports = function createFsAdapter(db) {
  const fs = {};

  fs.readFile = (filename, options, callback) => {
    if (typeof options === 'function') {
      callback = options;
      options = {};
    } else if(typeof options === 'string') {
      options = {encoding: options};
    }

    db.get(path.resolve(filename), {           //[1]
        valueEncoding: options.encoding
      },
      (err, value) => {
        if(err) {
          if(err.type === 'NotFoundError') {       //[2]
            err = new Error(`ENOENT, open "${filename}"`);
            err.code = 'ENOENT';
            err.errno = 34;
            err.path = filename;
          }
          return callback && callback(err);
        }
        callback && callback(null, value);       //[3]
      }
    );
  };

  fs.writeFile = (filename, contents, options, callback) => {
    if(typeof options === 'function') {
      callback = options;
      options = {};
    } else if(typeof options === 'string') {
      options = {encoding: options};
    }

    db.put(path.resolve(filename), contents, {
      valueEncoding: options.encoding
    }, callback);
  };

  return fs;
};
```

我们的新适配器插件已经准备就绪；如果我们现在编写一个测试模块，我们可以尝试使用它：

```
const fs = require('fs');

fs.writeFile('file.txt', 'Hello!', () => {
  fs.readFile('file.txt', {encoding: 'utf8'}, (err, res) => {
    console.log(res);
  });
});

// 试图读取不存在的文件
fs.readFile('missing.txt', {encoding: 'utf8'}, (err, res) => {
  console.log(err);
});
```

```
..06/15_adapter (zsh)
➜ 15_adapter git:(master) ✗ node testFs.js
{ Error: ENOENT: no such file or directory, open 'missing.txt'
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: 'missing.txt' }
Hello!
➜ 15_adapter git:(master) ✗
```

上面的代码使用原始的 `fs API` 在文件系统上执行一些读写操作，并应该在控制台上打印如下内容：

```
{ [Error: ENOENT, open 'missing.txt'] errno: 34, code: 'ENOENT',
 path: 'missing.txt' }
Hello!
```

现在，我们可以尝试用我们的适配器替换 `fs` 模块，如下所示：

```
const levelup = require('level');
const fsAdapter = require('./fsAdapter');
const db = levelup('./fsDB', {
  valueEncoding: 'binary'
});
const fs = fsAdapter(db);
```

再次运行我们的程序应该产生相同的输出，除了我们指定的文件没有任何部分是使用文件系统读取或写入的。相反，使用我们的适配器执行的任何操作都将转换为在 `LevelUP` 数据库上执行的操作。

我们刚创建的适配器可能看起来很傻，因为我们不明确使用数据库代替真正的文件系统的目的是什么。但是，我们应该记住， `LevelUP` 本身具有适配器，可以使数据库也在浏览器中运行；其中一个适配器是level.js。现在我们的适配器应该是完美的。我们可以考虑使用它来与依赖于 `fs` 模块的浏览器代码共享！例如，我们在 `Chapter 3-Asynchronous Control Flow Patterns with Callbacks` 中创建的 `Web爬虫应用程序` 使用 `fs API` 来存储在其操作期间下载的网页；我们的适配器将允许它在浏览器中运行只需稍作修改！我们很快就会意识到，在涉及到与浏览器共享代码时，适配器模式也是一个非常重要的模式，我们将在 `Chapter8-Universal JavaScript for Web Applications` 中详细介绍。

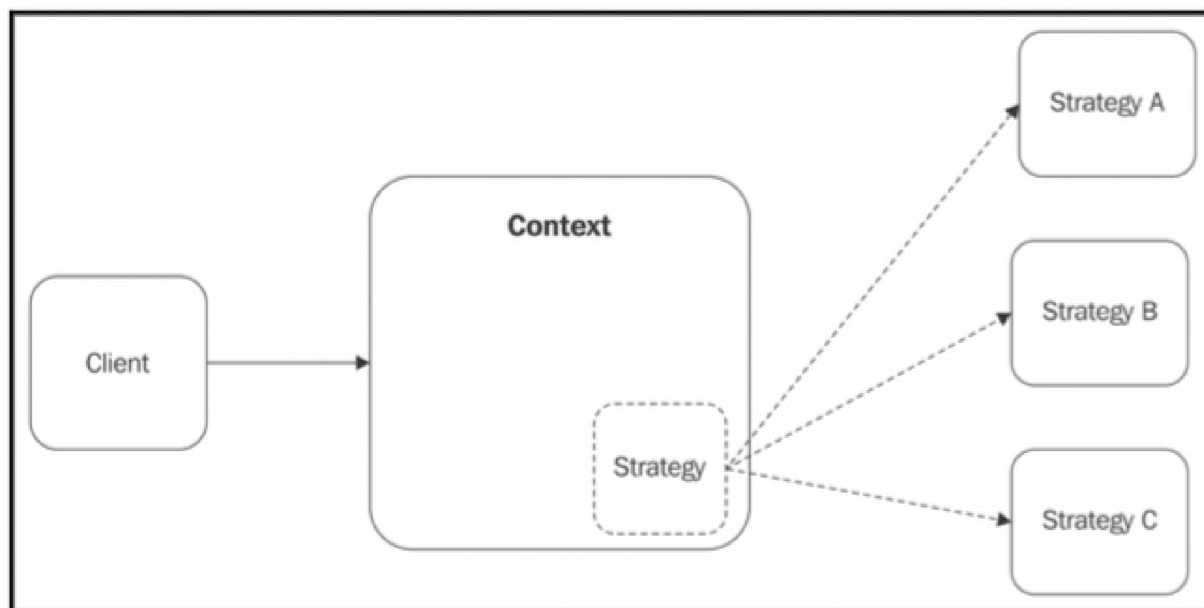
实际应用场景

适配器模式有很多实际应用场景的例子，在这里列出一些最值得注意的例子进行分析：

- 我们已经知道 `LevelUP` 能够在浏览器中使用不同的存储后端运行，从默认的 `LevelDB` 到 `IndexedDB` 。这是通过创建复制内部私有的 `LevelUP API` 的各种适配器实现的。可以到以下链接看看是如何实现的：https://github.com/rvagg/node-levelup/wiki/Modules#storage-back-ends 。
- `jugglingdb` 是一个多数据库的 `ORM` ，当然，使用多个适配器使其与不同的数据库兼容。可以到以下链接看看是如何实现的：https://github.com/1602/jugglingdb/tree/master/lib/adapters 。
- 对我们创建的例子的完美补充是level-filesystem，它是在 `LevelUP` 之上的 `fs API` 的正确实现。

策略模式（ `Strategy` ）

策略模式通过将可变部分提取为单独的，可交换的对象 `Strategy` 来使对象 `Context` 支持其逻辑中的变化。 `Context` 实现通用逻辑，而策略实现了可变部分，允许上下文根据不同因素（如输入值，系统配置或用户偏好）调整其行为。这些策略通常是解决方案的一部分，他们都实现了相同的接口，这是 `Context` 对象所期望的接口。下图显示了我们刚刚描述的情况：



上图显示了 `Context` 对象如何将不同的策略插入到其结构中，就好像它们是一个机器的可替换部分一样。想象一下汽车，其轮胎可以视为适应不同路况的策略。我们可以安装冬季轮胎在雪路上行驶，这要归功于他们的螺栓，而我们可以决定为高速公路行驶的高性能轮胎做长途旅行。一方面，我们不想把整个车改变，另一方面，我们不想要一辆八轮车，这样就可以在任何一条路上行驶。

我们很快就明白这种模式有多强大，不仅有助于分离算法中的关注点，而且还使其具有更好的灵活性并适应同一问题的不同变化。

策略模式在支持算法变化需要复杂的条件逻辑（大量的 `if...else` 或 `switch` 语句）或混合同一族不同算法的所有情况下特别有用。设想一个名为 `Order` 的对象，表示一个电子商务网站的在线订单。该对象有一个名为 `pay()` 的方法，就像它说的那样，完成订单并将资金从用户转移到商城用户。

为了支持不同的支付系统，我们有几个选项，如下所示：

- 在 `pay()` 方法中使用 `if...else` 语句来完成基于操作的操作。
- 在选择的付款选项上将支付的逻辑委托给实现用户选择的特定支付网关逻辑的策略对象。

在第一种解决方案中，我们的订单对象不能支持其他支付方式，除非其代码被修改。而且，当支付选项的数量增加时，这可能变得相当复杂。相反，使用策略模式使得 `Order` 对象支持几乎无限数量的支付方法，并且保持其范围仅限于管理用户的细节，购买的项目和相对价格，同时将完成支付的工作委派给另一个对象。

现在让我们用一个简单实际的例子来展示这个模式。

多格式配置对象

让我们考虑一个名为 `Config` 的对象，该对象包含应用程序使用的一组配置参数，例如数据库URL，服务器的侦听端口等。 `Config` 对象应该能够提供一个简单的接口来访问这些参数，而且还可以使用持久性存储（如文件）导入和导出配置。我们

希望能够支持不同的格式来存储配置，例如 `JSON` ， `INI` 或 `YAML` 。

通过应用我们了解的策略模式，我们可以立即识别 `Config` 对象的变量部分，这是允许我们序列化和反序列化配置的功能。

让我们创建一个名为 `config.js` 的新模块，让我们定义配置管理器的通用部分：

```
const fs = require('fs');
const objectPath = require('object-path');
class Config {
  constructor(strategy) {
    this.data = {};
    this.strategy = strategy;
  }
  get(path) {
    return objectPath.get(this.data, path);
  }
  // ...
}
```

在前面的代码中，我们将配置数据封装到一个实例变量（ `this.data` ）中，然后我们提供了 `set()` 和 `get()` 方法，允许我们使用 `object-path` 访问配置属性（例如， `property.subProperty` ），通过利用object-path。在构造函数中，我们也采取了一种策略作为输入，它代表解析和序列化数据的算法。

现在让我们看看我们将如何使用策略，开始编写 `Config` 类的剩余部分：

```
const fs = require('fs');
const objectPath = require('object-path');

class Config {
  constructor(strategy) {
    this.data = {};
    this.strategy = strategy;
  }

  get(path) {
    return objectPath.get(this.data, path);
  }

  set(path, value) {
    return objectPath.set(this.data, path, value);
  }

  read(file) {
    console.log(`Deserializing from ${file}`);
    this.data = this.strategy.deserialize(fs.readFileSync(file,
'utf-8'));
  }

  save(file) {
    console.log(`Serializing to ${file}`);
    fs.writeFileSync(file, this.strategy.serialize(this.data));
  }
}

module.exports = Config;
```

在前面的代码中，当从文件中读取配置时，我们将反序列化任务委托给策略；那么当我们想把配置保存到文件中时，我们使用策略来序列化配置。这个简单的设计允许 `Config` 对象在加载和保存数据时支持不同的文件格式。

为了演示这一点，我们在一个名为 `strategies.js` 的文件中创建一些策略。 让我们从解析和序列化 `JSON` 数据的策略开始：

```
module.exports.json = {
  deserialize: data => JSON.parse(data),
  serialize: data => JSON.stringify(data, null, '  ')
}
```

没有什么复杂的！我们的策略简单地实现了接口，以便它可以被 `Config` 对象使用。

同样，我们要创建的下一个策略允许我们支持 `INI` 文件格式：

```
const ini = require('ini'); // https://npmjs.org/package/ini
module.exports.ini = {
  deserialize: data => ini.parse(data),
  serialize: data => ini.stringify(data)
}
```

现在，为了向您展示如何结合在一起，我们创建一个名为 `configTest.js` 的文件，让我们尝试使用不同的格式文件加载和保存示例配置：

```
const Config = require('./config');
const strategies = require('./strategies');
const jsonConfig = new Config(strategies.json);
jsonConfig.read('samples/conf.json');
jsonConfig.set('book.nodejs', 'design patterns');
jsonConfig.save('samples/conf_mod.json');
const iniConfig = new Config(strategies.ini);
iniConfig.read('samples/conf.ini');
iniConfig.set('book.nodejs', 'design patterns');
iniConfig.save('samples/conf_mod.ini');
```

我们的测试模块揭示了策略模式的属性。我们只定义了一个 `Config` 类，它实现了我们的配置管理器的公共部分，同时改变了用于序列化和反序列化的策略，允许我们创建支持不同文件格式的不同 `Config` 实例。

前面的例子只显示了使用策略模式实现多格式配置对象的方法之一。其他有效的方法可能如下：

- 创建两个不同的策略系列：一个用于反序列化，另一个用于序列化。这将允许从格式读取并保存到另一个格式。
- 根据所提供文件的扩展名，动态选择策略； `Config` 对象可以保持一个 `map extension` -> `strategy` ，并用它来为给定的扩展名选择正确的算法。

正如我们所看到的，有几种选择使用策略的选择，正确的选择取决于我们的要求，以及我们希望获得的特性/简单性的折衷。

而且，模式本身的实现可能会有很大的不同，例如，以其最简单的形式， `context` 和 `strategy` 都可以是简单的函数：

```
function context(strategy) {...}
```

尽管前面的情况看起来可能微不足道，但在 `JavaScript` 等编程语言中，函数是一等公民，并且可以用作完全成熟的对象。

在所有这些变化之间，不变的是模式背后的思想；模式的实现可以稍微改变，但驱动模式实现的核心概念永远是一样的。

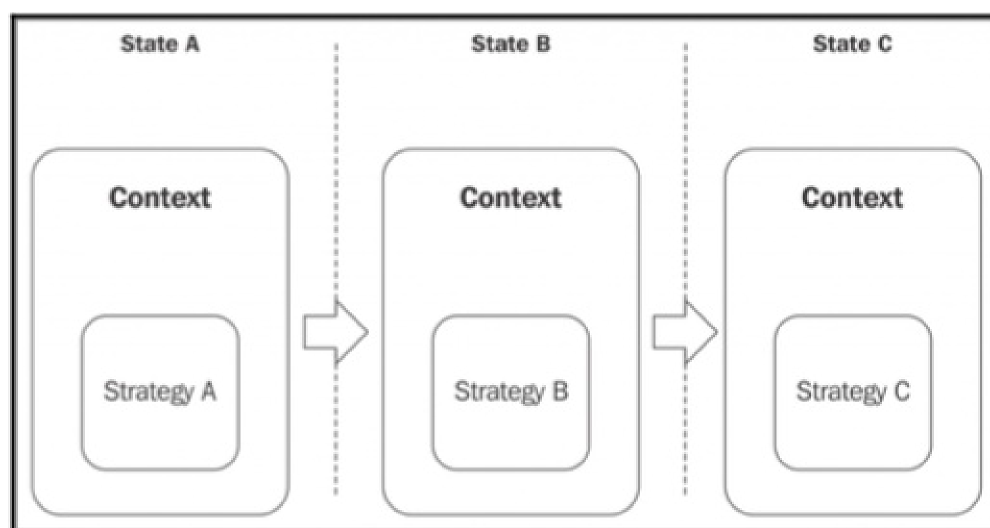
实际应用场景

Passport.js是 `Node.js` 的认证框架，它允许在 `Web` 服务器上支持不同的认证方案。通过 `Passport` ，我们可以轻松使用 `Facebook` 登录或使用 `Twitter` 登录功能到我们的 `Web` 应用程序。 `Passport` 使用策略模式将认证过程中所需的公共逻辑与可以更改的部分（即实际的认证步骤）分开。例如，我们可能想要使用 `OAuth` 来获取访问令牌来访问 `Facebook` 或 `Twitter` 个人资料，或者只需使用本地数据库来验证用户名/密码。对于 `Passport` ，这些都是完成身份验证过程的不同策略，正如我们所能想象的，这使得这个库可以支持几乎无限的身份验证服务。客户以看看 http://passportjs.org/guide/providers 上支持的不同身份验证，以了解策略模式可以执行的操作。

状态模式（ `State` ）

状态模式是策略模式的变体，策略根据 `Context` 的状态而变化。 我们在前面的章节已经看到，如何根据用户的偏好，配置参数和提供的输入等不同的变量来选择一个策略，一旦这个选择完成，策略在 `Context` 剩余的寿命期间保持不变。

相反，在状态模式中，策略（在这种情况下也称为状态）是动态的，可以在 `Context` 的生命周期中改变，从而允许其行为根据其内部状态进行调整，如下图所示：



想象一下，我们有一个酒店预订系统和一个 `Reservation` 对象来模拟房间预订。

这是一个经典的情况，我们必须根据其状态来调整对象的行为。考虑以下一系列事件：

1. 当订单初始创建时，用户可以使用 `confirm()` 方法确认订单；当然，他们不能使用 `cancel()` 方法取消预约，因为订单还没有被确认。但是，如果他们在购买之前改变主意，他们可以使用 `delete()` 方法删除它。
2. 一旦确认订单，再次使用 `confirm()` 方法没有任何意义；不过，现在应该可以取消预约，但不能再删除，因为要保留对应记录。

3. 在预约日期前一天，不应取消订单。因为这太迟了。

现在想象一下，我们必须实现我们在一个单一的对象中描述的预订系统；我们已经可以画出所有的 `if...else` 或者 `switch` 语句逻辑图，这些语句是我们必须写的，以便根据预留的状态来启用/禁用每个动作。

在这种情况下，状态模式是完美的：将会有三种策略，全部实现描述的三个方法（ `confirm()` ， `cancel()` 和 `delete()` ），每个只执行一个行为，一个策略对应于一种状态。通过使用状态模式， `Reservation` 对象从一个行为切换到另一个行为应该是非常容易的。这只需要在每个状态变化上激活一个不同的策略。

状态转换可以由 `Context` 对象，客户端代码或 `State` 对象本身启动和控制。通常由 `State` 对象本身控制，因为这在灵活性和解耦方面效果较好，因为 `Context` 对象不必知道所有可能的状态以及如何在它们之间转换。

实现一个基本的 `fail-safe socket`

现在我们来看一个具体的例子，以便我们能够运用我们所了解到的状态模式。让我们建立一个客户端 `TCP` 套接字，当与服务器的连接丢失时不会丢失客户端请求；相反，我们希望将服务器处于脱机状态的时间内发送的所有数据进行排队，然后在连接重新建立后立即尝试发送。我们希望在一个简单的监控系统中利用这个套接字，在这个系统中，一组机器每隔一段时间发送一些关于资源利用率的统计信息；如果收集这些资源的服务器关闭，则我们的套接字将继续在本地排队数据，直到服务器重新联机为止。

首先创建一个名为 `failsafeSocket.js` 的模块来表示我们的 `context` 对象：

```
const OfflineState = require('./offlineState');
const OnlineState = require('./onlineState');

class FailsafeSocket {
  constructor (options) { // [1]
    this.options = options;
    this.queue = [];
    this.currentState = null;
    this.socket = null;
    this.states = {
      offline: new OfflineState(this),
      online: new OnlineState(this)
    };
    this.changeState('offline');
  }

  changeState (state) { // [2]
    console.log('Activating state: ' + state);
    this.currentState = this.states[state];
    this.currentState.activate();
  }

  send(data) { // [3]
    this.currentState.send(data);
  }
}

module.exports = options => {
  return new FailsafeSocket(options);
};
```

`FailsafeSocket` 类由三个主要元素组成：

1. 构造函数初始化各种数据结构，包括将包含在套接字脱机时发送的任何数据的队列。此外，它还创建了一组两个状态，一个用于在脱机状态下实现套接字的行为，另一个用于在套接字处于联机状态时的状态。
2. `changeState()` 方法负责从一个状态转换到另一个状态。 它只是更新 `currentState` 实例变量，并调用目标状态的 `activate()` 。
3. `send()` 方法是套接字的功能，这是我们希望基于离线/在线状态具有不同行为的地方。我们可以看到，这是通过将操作委托给当前活动状态来完成的。

现在让我们来看看这两个状态是什么样子的，从 `offlineState.js` 模块开始：

```
const jot = require('json-over-tcp'); // [1]

module.exports = class OfflineState {

  constructor (failsafeSocket) {
    this.failsafeSocket = failsafeSocket;
  }

  send(data) { // [2]
    this.failsafeSocket.queue.push(data);
  }

  activate() { // [3]
    const retry = () => {
      setTimeout(() => this.activate(), 500);
    };

    this.failsafeSocket.socket = jot.connect(
      this.failsafeSocket.options,
      () => {
        this.failsafeSocket.socket.removeListener('error', retry
);
        this.failsafeSocket.changeState('online');
      }
    );
    this.failsafeSocket.socket.once('error', retry);
  }
};
```

我们创建的模块负责在脱机状态下管理套接字的行为：

1. 我们将使用一个名为json-over-tcp的库来代替使用原始的 `TCP` 套接字，这将使我们能够轻松地在一个 `TCP` 连接中发送 `JSON` 对象。
2. `send()` 方法只负责排队它接收到的任何数据。我们假设我们是离线的，这就是我们需要做的。
3. `activate()` 方法尝试使用 `json-over-tcp` 与服务器建立连接。如果操作失败，则在 `500` 毫秒后再次尝试。它会继续尝试，直到建立有效的连接，在这种情况下， `failafeSocket` 的状态将转换为联机状态。

接下来，让我们实现 `onlineState.js` 模块，然后让我们实现 `onlineState` 策略，如下所示：

```
module.exports = class OnlineState {
  constructor(failsafeSocket) {
    this.failsafeSocket = failsafeSocket;
  }

  send(data) { // [1]
    this.failsafeSocket.socket.write(data);
  };

  activate() { // [2]
    this.failsafeSocket.queue.forEach(data => {
      this.failsafeSocket.socket.write(data);
    });
    this.failsafeSocket.queue = [];

    this.failsafeSocket.socket.once('error', () => {
      this.failsafeSocket.changeState('offline');
    });
  }
};
```

`OnlineState` 策略非常简单，解释如下：

1. `send()` 方法直接将数据写入套接字，因为我们假设 `TCP` 已连接。
2. `activate()` 方法刷新套接字处于脱机状态时排队的所有数据，并且还开始监听任何 `error` 事件；我们将把这个作为套接字下线的前兆。发生这种情况时，我们转换到 `offline` 状态。

这就是 `failsafeSocket` ；现在我们准备构建一个示例客户端和一个服务器来尝试。把服务器代码放在一个名为 `server.js` 的模块中：

```
const jot = require('json-over-tcp');
const server = jot.createServer({
  port: 5000
});
server.on('connection', socket => {
  socket.on('data', data => {
    console.log('Client data', data);
  });
});

server.listen({
  port: 5000
}, () => console.log('Started'));
```

> 注意：原书的代码有错，现在的 `jot.createServer()` 接受的参数是一个对象，这里把书上的 `5000` 改为 `{ post: 5000 }` 。

然后看客户端代码 `client.js` ：

```
const createFailsafeSocket = require('./failsafeSocket');
const failsafeSocket = createFailsafeSocket({
  port: 5000
});
setInterval(() => {
  // 每隔1000毫秒发送当前内存使用状态
  failsafeSocket.send(process.memoryUsage());
}, 1000);
```

```
node (node)
➜ 17_state git:(master) ✗ node client.js
Activating state: offline
Activating state: online
Activating state: offline
Activating state: online
Activating state: offline
```

```
..er06/17_state (zsh)
➜ 17_state git:(master) ✗ node server.js
Started
Client data { rss: 22130688,
  heapTotal: 7684096,
  heapUsed: 4839992,
  external: 20038 }
Client data { rss: 22183936,
  heapTotal: 7684096,
  heapUsed: 4855496,
  external: 20116 }
Client data { rss: 22188032,
  heapTotal: 7684096,
  heapUsed: 4859760,
  external: 20194 }
Client data { rss: 22188032,
  heapTotal: 7684096,
  heapUsed: 4861544,
  external: 20272 }
^C
➜ 17_state git:(master) ✗ node server.js
Started
Client data { rss: 22278144,
  heapTotal: 7684096,
  heapUsed: 4942240,
```

我们的服务器只是打印它接收到的任何 `JSON` 对象消息给控制台，而我们的客户端利用一个 `FailsafeSocket` 对象每秒发送一次内存利用率的测量值。

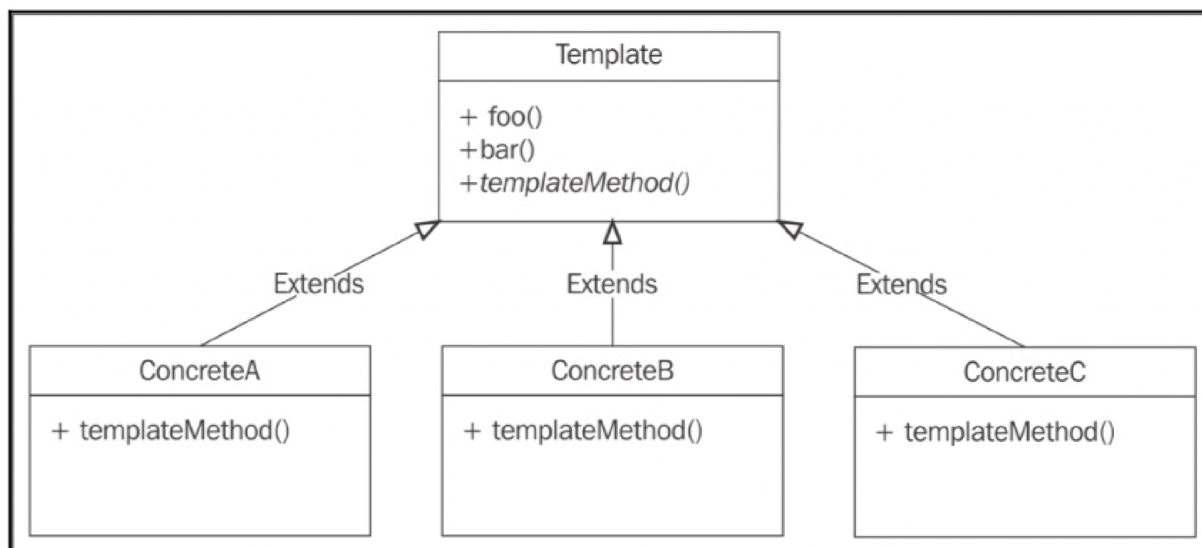
尝试构建的小型系统，我们应该运行客户端和服务器，然后通过停止然后重新启动服务器来测试 `failafeSocket` 的功能。 我们应该看到，客户端的状态在线和离线之间发生了变化，服务器离线时收集的任何请求都会排队，然后在服务器重新联机后重新发送。

这个例子应该清楚地说明状态模式如何能够帮助增加一个组件的模块化和可读性，这个组件必须根据状态来调整它的行为。

> 我们在本节中构建的 `FailsafeSocket` 类仅用于演示状态模式，并不希望成为处理 `TCP` 套接字内连接问题的完整且 `100%` 可靠的解决方案。例如，我们不验证写入套接字流，而让所有数据都被服务器接收到，这将需要更多与我们想描述的模式无关的代码。

模板模式（`Template`）

我们将要分析的下一个模式叫做模板模式，它与策略模式有许多共同点。模板由定义一个抽象的伪类组成，它代表了算法的框架，其中一些步骤是未定义的。然后子类可以通过实现缺少的步骤填充算法中的空白，称为模板方法。这种模式的目的是使定义一个类的家族成为可能，这些类都是类似算法的变体。下面的 `UML` 图显示了我们刚刚描述的结构：



上图中显示的三个具体类扩展了 `Template` 并为 `templateMethod()` 提供了一个实现，使用 `C++` 术语来说，该实现方法是抽象或者说是虚函数；在 `JavaScript` 中，这意味着该方法是未定义的或被分配给一个总是抛出异常的函数，这表明该方法必须被实现。模板模式可以被认为比我们目前所看到的其他模式更加符合面向对象思想，因为继承是其实现的核心部分。

模板模式和策略模式的目的非常相似，但两者的主要区别在于它们的结构和实现。两者都允许我们改变算法的某些部分，同时重用公共部分；然而，尽管策略模式允许我们在运行时动态地执行它，但使用模板模式完成算法是在具体类被定义的时候确定的。在这些假设下，模板模式可能更适合那些我们想要创建一个算法的预先打包的变体的情况。与往常一样，一种模式与另一种模式的选择取决于开发者，他们必须考虑每个用例的各种利弊。

使用模板模式的配置管理器

为了更好地了解模板模式和状态模式之间的区别，现在让我们重新实现我们在关于策略模式的章节中定义的 `Config` 对象，但是这次使用模板模式。就像以前版本的 `Config` 对象一样，我们希望能够使用不同的文件格式来加载和保存一组配置属性。

首先定义模板类，我们将其称为 `ConfigTemplate` ：

```
const fs = require('fs');
const objectPath = require('object-path');
class ConfigTemplate {
  read(file) {
    console.log(`Deserializing from ${file}`);
    this.data = this._deserialize(fs.readFileSync(file, 'utf-8')
);
  }
  save(file) {
    console.log(`Serializing to ${file}`);
    fs.writeFileSync(file, this._serialize(this.data));
  }
  get(path) {
    return objectPath.get(this.data, path);
  }
  set(path, value) {
    return objectPath.set(this.data, path, value);
  }
  _serialize() {
    throw new Error('_serialize() must be implemented');
  }
  _deserialize() {
    throw new Error('_deserialize() must be implemented');
  }
}
module.exports = ConfigTemplate;
```

新的 `ConfigTemplate` 类定义了两个模板方法：`_deserialize()` 和 `_serialize()`，它们是执行加载和保存配置所需的。名称开头的下划线表示它们仅供内部使用，这是一种标记受保护方法的简单方法。由于在 `JavaScript` 中我们不能将方法声明为抽象方法，我们简单地将它们定义为存根，如果它们被调用（即，如果它们没有被具体子类覆盖）则抛出异常。

现在让我们使用我们的模板创建一个具体的类，例如，允许我们使用 `JSON` 格式加载和保存配置：

```
const util = require('util');
const ConfigTemplate = require('./configTemplate');
class JsonConfig extends ConfigTemplate {
  _deserialize(data) {
    return JSON.parse(data);
  };
  _serialize(data) {
    return JSON.stringify(data, null, '  ');
  }
}
module.exports = JsonConfig;
```

`JsonConfig` 类从我们的模板， `ConfigTemplate` 类和
为 `_deserialize()` 和 `_serialize()` 方法提供了一个具体的实
现。 `JsonConfig` 类现在可以作为独立的配置对象使用，而不使用 需要指定一个
序列化和反序列化的策略，因为它是在类本身中实现的：

```
const JsonConfig = require('./jsonConfig');

const jsonConfig = new JsonConfig();
jsonConfig.read('samples/conf.json');
jsonConfig.set('nodejs', 'design patterns');
jsonConfig.save('samples/conf_mod.json');
```

```
..6/18_template
➜ 18_template git:(master) ✗ node configTest.js
Deserializing from samples/conf.json
Serializing to samples/conf_mod.json
➜ 18_template git:(master) ✗ cat samples/conf.json
{
  "property": "test"
}
➜ 18_template git:(master) ✗ cat samples/conf_mod.json
{
  "property": "test",
  "nodejs": "design patterns"
}%
➜ 18_template git:(master) ✗
```

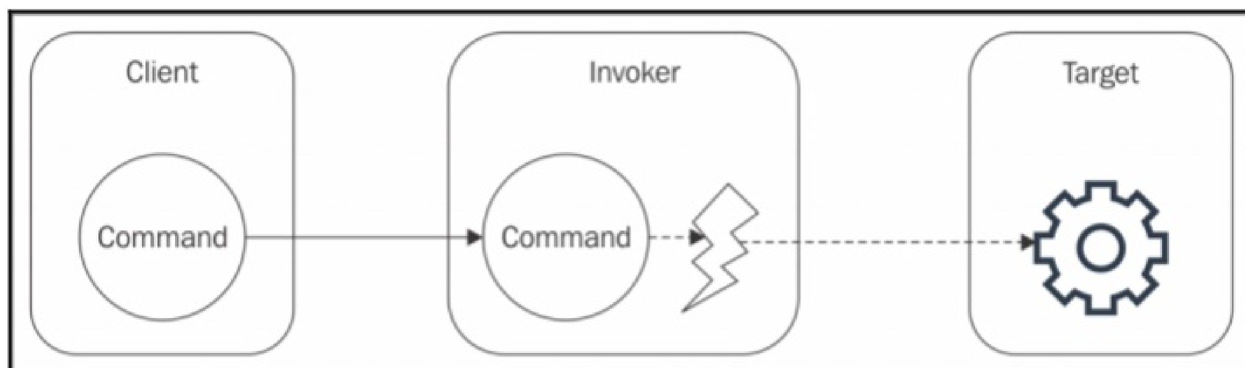
通过使用模板模式，我们可以通过重复使用从父模板类继承的逻辑和接口，仅提供一些抽象方法的实现，从而使我们能够获得一个全新的完全配置管理器。

实际应用场景

这种模式不应该听起来对我们来说是全新的。我们已经
在 `Chapter 5-Coding with Streams` 时遇到过它，当我们扩展不同
的 `Streams` 类来实现我们的自定义流。在这种情况下，模板方法
是 `_write()` ， `_read()` ， `_transform()` 或 `_flush()` 方法，具体取决于我们想要实现的流类。要创建一个新的自定义流，我们需要从一个特定的抽象流类继承，为模板方法提供一个实现。

命令模式（`Command`）

命令模式是在 `Node.js` 中另一个重要的设计模式。在其最通用的定义中，命令模式封装了主体对象信息，并对主体对象执行一个动作，而不是在主体对象上直接调用一个方法或一个函数，我们创建一个对象 `invocation` 执行这样一个 调用；那么实现这个意图将是另一个组件的责任，将其转化为实际行动。传统上，这个模式是围绕着四个主要的组件，如下图所示：



命令模式的典型组织可以描述如下：

- `Command` ：这是封装调用一个必要信息的对象方法或功能。
- `Client` ：这将创建该命令并将其提供给调用者。
- `Invoker` ：这是负责执行目标上的命令。
- `Target` （或 `Receiver` ）：这是调用的主题。它可以是一个单独的功能或对象的方法。

正如我们将看到的，这四个组件可以根据我们想要的方式变化很多实施模式；在这一点上，这听起来不是什么新鲜事。使用命令模式而不是直接执行一个操作有好几个。

优点和应用：

- 命令可以安排在稍后执行。
- 一个命令可以很容易地序列化并通过网络发送。这很简单，属性允许我们在远程机器上分配作业，传输命令
- 从浏览器到服务器，创建 `RPC` 系统等等。
- 通过命令可以很容易地在系统上保存所有执行的操作历史记录。
- 命令是一些数据同步算法的重要组成部分和解决冲突。
- 计划执行的命令如果尚未执行，则可以取消。它 也可以恢复（撤消），使应用程序的状态的重点 在命令执行之前。
- 几个命令可以组合在一起。这可以用来创建原子交易或实施一个机制，从而在所有的操作组立即执行。
- 可以对一组命令执行不同类型的转换，例如 作为重复删除，加入和拆分，或应用更复杂的算法如 `Operational Transformation` (`OT`)，这是当今大多数的基础实时协作软件，如协同文本编辑。

前面的列表清楚地向我们展示了这种模式的重要性，特别是在 `node.js` 这样的平台中，网络和异步执行是必不可少的参与者。

灵活模式

正如我们已经提到的，`JavaScript` 中的命令模式可以通过许多不同的方式实现；我们现在只演示其中的几个，只是为了给出它的范围的概念。

任务模式

我们可以从最基本的和平凡的实现开始：任务模式。当然，`JavaScript` 中创建一个表示调用的对象的最简单方法是创建一个关闭：

```
function createTask(target, args) {
  return () => {
    target.apply(null, args);
  }
}
```

这看起来一点也不新鲜；我们已经在书中多次使用了这种模式，特别是在第3章，带有回调的异步控制流模式中。这种技术允许我们使用单独的组件来控制和调度任务的执行，这在本质上等同于命令模式的调用者。例如，您还记得我们是如何定义传递给异步库的任务的吗？或者更好的是，你还记得我们是如何结合使用发电机的吗？回调模式本身可以被认为是命令模式的一个非常简单的版本。

较复杂的命令模式

现在让我们来处理一个更复杂的命令的示例；这一次我们希望支持撤消和序列化。让我们从命令的目标开始，这个小对象负责向Twitter这样的服务发送状态更新。为了简单起见，我们使用这种服务的模拟：

```
const statusUpdateService = {
  statusUpdates: {},
  sendUpdate: function(status) {
    console.log('Status sent: ' + status);
    Design Patterns
      [252]
    let id = Math.floor(Math.random() * 1000000);
    statusUpdateService.statusUpdates[id] = status;
    return id;
  },
  destroyUpdate: id => {
    console.log('Status removed: ' + id);
    delete statusUpdateService.statusUpdates[id];
  }
};
```

现在，让我们创建一个命令来表示新状态更新的发布：

```
function createSendStatusCmd(service, status) {
  let postId = null;
  const command = () => {
    postId = service.sendUpdate(status);
  };
  command.undo = () => {
    if (postId) {
      service.destroyUpdate(postId);
      postId = null;
    }
  };
  command.serialize = () => {
    return {
      type: 'status',
      action: 'post',
      status: status
    };
  };
  return command;
}
```

前面的函数是一个工厂，它生成新的 `sendstate` 命令。每个命令实现以下三个功能：

1. 命令本身是一个函数，当调用它时，它将触发操作；换句话说，它实现了我们前面看到的任务模式。该命令在执行时将使用目标服务的方法发送新的状态更新。
2. 连接到主任务的 `auto()` 函数，该函数恢复操作的效果。在我们的例子中，我们只是调用目标服务上的 `deadyupdate()` 方法。
3. `serialize()` 函数，它构建一个 `json` 对象，该对象包含重建同一个命令对象所需的所有信息。 在此之后，我们可以构建一个调用程序；我们可以通过实现它的构造函数和它的 `run()` 方法来开始：

```
class Invoker {
constructor() {
 this.history = [];
}
run(cmd) {
 this.history.push(cmd);
 cmd();
 console.log('Command executed', cmd.serialize());
}
}
```

前面定义的 `run()` 方法是 `Invoker` 的基本功能；它负责将命令保存到 `history` 实例变量中，然后触发命令本身的执行。接下来，我们可以添加一个延迟执行命令的新方法：

```
delay(cmd, delay) {
setTimeout(() => {
 this.run(cmd);
}, delay)
}
```

然后，我们可以实现一个 `undo()` 方法来恢复最后一个命令：

```
undo() {
const cmd = this.history.pop();
cmd.undo();
console.log('Command undone', cmd.serialize());
}
```

最后，我们还希望能够在远程服务器上运行命令，方法是使用 `Web` 服务序列化并通过网络传输命令：

```
runRemotely(cmd) {
request.post('http://localhost:3000/cmd', {
   json: cmd.serialize()
 },
 err => {
   console.log('Command executed remotely', cmd.serialize())
;
 }
);
}
```

既然我们有了命令、调用程序和目标，唯一缺少的组件就是客户端。让我们从实例化 `Invoker` 开始：

```
const invoker = new Invoker();
```

然后，我们可以使用以下代码行创建一个命令：

```
const command = createSendStatusCmd(statusUpdateService, 'HI
!');
```

现在我们有了一个命令，表示状态消息的发布；然后我们可以决定立即发送它：

```
invoker.run(command);
```

但是，我们犯了一个错误；让我们恢复到时间线的状态，就像发送最后一条消息之前的情况一样：

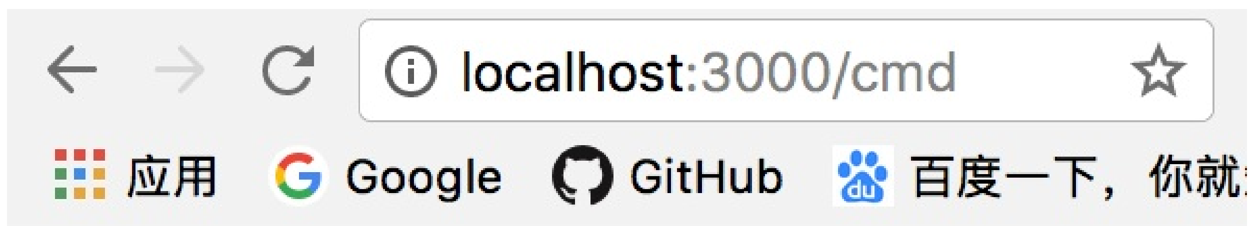
```
invoker.undo();
```

我们还可以决定从现在起一小时内发送消息：

```
invoker.delay(command, 1000 * 60 * 60);
```

或者，我们可以通过将任务迁移到另一台机器来分配应用程序的负载：

```
invoker.runRemotely(command);
```




```
{
    ok: true
}
```

我们刚刚创建的一个小例子展示了如何在命令中包装一个操作可以打开一个可能性的世界，这只是冰山一角。

正如最后的讨论，值得注意的是，只有在真正需要的时候才会使用成熟的命令模式。事实上，我们看到了我们需要编写多少额外的代码来简单地调用 `statuupdatesservice` 方法；如果我们所需要的只是一个调用，那么一个复杂的命令就会被杀死。但是，如果我们需要安排任务的执行，或者运行异步操作，那么简单的任务模式提供了最好的折衷。如果相反，我们需要更高级的特性，如撤销支持、转换、冲突解决，或者我们前面描述的其他花哨用例之一，那么对命令使用更复杂的表示几乎是必要的。

中间件模式（ `Middleware` ）

`Node.js` 中最有特色的模式之一绝对是中间件模式。不幸的是，对于没有经验的人来说，这也是最令人困惑的事情之一，特别是来自企业架构的开发人员。疑惑的原因可能与中间件这个术语的含义有关，中间件在企业架构术语中表示各种软件套件，这些软件套件有助于抽象 `OS API` ， `网络通信` ， `内存管理` 等较底层的操作，允许开发人员只关注应用程序的商业案例。在这种情况下，中间件回顾了诸

如 `CORBA` ， `Enterprise Service Bus` ， `Spring` ， `JBoss` 等主题，但是在更通用的意义上，它也可以定义任何类型的软件层，它们在低级服务和应用程序字面上是中间的软件）。

`Express` 的中间件

在 `Node.js` 中，Express广泛使用中间件模式。在 `Express` 中，事实上，中间件表示一组服务，通常是函数，它们被组织在一个 `pipeline` 中，负责处理传入的 `HTTP` 请求和进行响应。

`Express` 是一个非常独特和简约的网络框架。使用中间件模式是一种有效的策略，它允许开发人员轻松创建、分发、添加新功能到当前应用程序。

`Express` 中间件是以下形式：

```
function(req, res, next) { ... }
```

在这里， `req` 是传入的 `HTTP` 请求， `res` 是响应， `next` 是当前中间件完成其任务时调用的回调，用来触发 `pipeline` 中的下一个中间件。 `Express` 中间件执行的任务包括以下内容：

- 解析请求的 `body`
- 压缩/解压 `req` 和 `res` 对象
- 生成访问日志
- 管理 `sessions`
- 管理加密的 `cookie`
- 提供跨站请求伪造（ `CSRF` ）保护

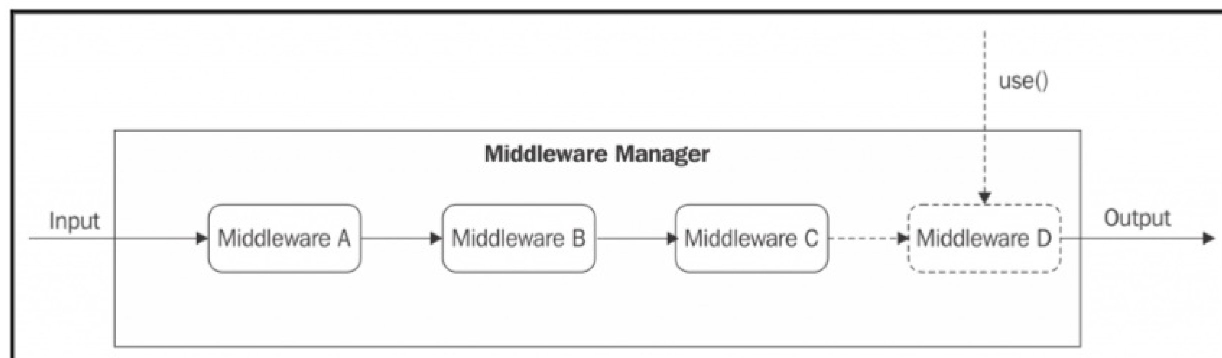
这些都是与应用程序的主要业务逻辑没有严格关联的任务，也不是 `Web` 服务器最核心的部分；它们是应用程序公共功能的中间件，使得实际的请求处理程序只关注其主要业务逻辑。从本质上讲，这些公共中间件是很有必要的。

中间件的模式

在 `Express` 中实现中间件的技术并不新鲜，实际上，它可以被看作是拦截过滤器模式和责任链模式的 `Node.js` 版本。用更一般的术语来说，它也代表了一个 `pipeline` 。现在的 `Node.js` 中，中间件这个术语不只是在 `Express` 框架中广泛使用，而是代表着一种特殊的模式，即一组处理单元，过滤器和处理程序以函数的形式连接起来形成一个异步序列，这个异步序列可以对任何类型数据进行预处理和后处理。这种模式的主要优点是灵活性；实际上，这种模式使我们能够以极低的代价生成 `Node.js` 基础架构，对于添加应用程序拓展和插件上提供了一种便捷灵活的方式。

> 如果您想了解更多关于拦截过滤器模式，可以阅读下面这篇文章：http://www.oracle.com/technetwork/java/interceptingfilter-142169.html ，这篇文章也很好地讲述了责任链模式： http://java.dzone.com/articles/design-patterns-uncovered-chain-of-responsibility

下图显示了中间件模式的组件：



该模式的基本组成部分是中间件管理器，负责组织和执行中间件功能。模式最重要的实现细节如下：

- 新的中间件可以通过调用 `use()` 函数来注册（这个函数的名字在这个模式的许多实现中是一个常见的约定，但我们可以选择任何名字）。通常情况下，新的中间件只能附加在 `pipeline` 的末尾，但这不是一个严格的规则。
- 当接收到新数据进行处理时，注册的中间件在异步顺序执行流程中被调用。 `pipeline` 中的每个单元接收前一个单元的执行结果作为输入。
- 每个中间件都可以通过简单地不调用回调或者向回调传递错误来决定停止进一步处理数据。错误情况通常会触发执行另一个专门用于处理错误的中间件序列。

数据如何在 `pipeline` 中处理和传输没有严格的规定。一般说来处理数据的方式有以下几点：

- 为结果数据增加额外的属性或方法，用于拓展数据
- 用某种处理的结果替换结果数据
- 保持数据不变，但总是返回处理结果的副本

如何选取中间件在 `pipeline` 中传输的策略，取决于中间件管理器的实现方式以及中间件本身执行的处理类型。

为 `ØMQ` 创建一个中间件框架

现在让我们通过围绕ØMQ消息传递库构建一个中间件框架来演示中间件模式。 `ØMQ` （也称为 `ZMQ` 或 `ZeroMQ` ）提供了一个简单的接口，用于通过各种协议在网络中交换原子消息；它的性能绝佳，其基本的抽象集是专门构建的，以促进自定义消息体系结构的实现。因此，经常选择 `ØMQ` 来构建复杂的分布式系统。

> 在 `Chapter11-Messaging and Integration Patterns` ，我们将有机会更详细地分析 `ØMQ` 的特性。

ØMQ的接口相当低级；它只允许我们为消息使用字符串和二进制缓冲区，所以任何编码或数据的自定义格式都必须由库的用户来实现。

在下一个示例中，我们将构建一个中间件基础结构，以抽象通过 `ØMQ` 套接字传递的数据的预处理和后处理，以便我们可以透明地处理JSON对象，同时无缝地压缩通过线路传递的消息。

> 在继续该示例之前，请确保按照此 `URL` 的说明安装 `ØMQ` 库：http://zeromq.org/intro:get-the-software 。4.0以上任何版本都应该足够用于这个例子。

中间件管理器

围绕 `ØMQ` 构建中间件基础架构的第一步是创建一个组件，负责在中间件管道中处理收到的消息和发送新消息。为此，我们创建一个名为 `zmqMiddlewareManager.js` 的新模块，并如下定义它：

```
module.exports = class ZmqMiddlewareManager {
  constructor(socket) {
    this.socket = socket;
    this.inboundMiddleware = []; // [1]
    this.outboundMiddleware = [];
    socket.on('message', message => { // [2]
      this.executeMiddleware(this.inboundMiddleware, {
        data: message
      });
    });
  }

  send(data) {
    const message = {
      data: data
    };

    this.executeMiddleware(this.outboundMiddleware, message,
      () => {
        this.socket.send(message.data);
      }
    );
  }

  use(middleware) {
    if (middleware.inbound) {
      this.inboundMiddleware.push(middleware.inbound);
    }
    if (middleware.outbound) {
      this.outboundMiddleware.unshift(middleware.outbound);
    }
  }

  executeMiddleware(middleware, arg, finish) {
    function iterator(index) {
      if (index === middleware.length) {
        return finish && finish();
      }
      middleware[index].call(this, arg, err => {
        if (err) {
          return console.log('There was an error: ' + err.messag
e);
        }
        iterator.call(this, ++index);
      });
    }

    iterator.call(this, 0);
  }
};
```

在这个类的第一部分，我们定义了这个新组件的构造函数。 它接受一个 `ØMQ` 套接字作为参数，并且：

1. 创建两个包含我们的中间件函数的空列表，一个用于入站消息，另一个用于出站消息。
2. 通过将一个新的监听器附加到 `message` 事件，它立即开始监听来自套接字的新消息。在侦听器中，我们通过执行 `inboundMiddleware` 管道来处理入站消息。

`ZmqMiddlewareManager` 类的下一个方法 `send` 负责在通过套接字发送新消息时执行中间件。

这次使用 `outboundMiddleware` 列表中的过滤器处理消息，然后将其传递给 `socket.send()` 以用于实际的网络传输。

现在，我们来谈谈 `use()` 方法。这个方法对于将新的中间件功能添加到我们的管道。每个中间件都是成对的；在我们的实现中，它是一个包含 `inbound` 和 `outbound` 两个属性的对象，这些属性则是要添加到相应列表的中间件函数。

在这里观察到， `inbound` 中间件被 `push` 到 `inboundMiddleware` 列表的末尾，而对于 `outboundMiddleware` 列表，则使用 `unshift` 在开始处插入 `outbound` 中间件。这是因为 `inbound / outbound` 中间件函数通常需要以相反的顺序执行。例如，如果我们想要使用 `JSON` 解压缩并反序列化 `inbound` 消息，则意味着对于 `outbound` ，我们应该首先序列化并压缩。

> 理解这个用于组织中间件的约定不是一般模式的一部分，而只是我们具体例子的一个实现细节。

最后一个函数 `executeMiddleware` 代表了我们组件的核心，它是负责执行中间件功能的函数。这个函数的代码应该看起来很熟悉， 实际上，它是我们在 `Chapter3-Asynchronous Control Flow Patterns with Callbacks` 中学习的异步顺序迭代模式的简单实现。作为输入接收的中间件队列中的每个函数被一个接一个地执行，并且为每个中间件功能提供相同的 `arg` 对象作为参数；这是可以将数据从一个中间件传播到下一个中间件的技巧。在迭代结束时，调用 `finish()` 回调。

> 为了简洁，我们不支持 `error` 中间件管道。 通常，当中间件功能传播错误时，执行专门用于处理错误的另一组中间件。这可以使用我们在这里演示的相同技术轻松实现。

支持 `JSON` 消息的中间件

现在我们已经实现了中间件管理器，我们可以创建一对中间件函数来演示如何处理 `inbound` 和 `outbound` 消息。正如我们所说的，我们的中间件基础架构的目标之一就是拥有一个过滤器来对 `JSON` 消息进行序列化和反序列化，所以让我们来创建新的中间件来处理这个问题。在一个名为 `jsonMiddleware.js` 的新模块中，我们包含以下代码：

```
module.exports.json = () => {
  return {
    inbound: function(message, next) {
      message.data = JSON.parse(message.data.toString());
      next();
    },
    outbound: function(message, next) {
      message.data = new Buffer(JSON.stringify(message.data));
      next();
    }
  }
};
```

我们刚刚创建的 `json` 中间件非常简单：

- `inbound` 中间件将收到的消息反序列化为输入，并将结果返回给消息的 `data` 属性，以便可以沿管道进一步处理
- `outbound` 中间件序列化 `message.data` 中的任何数据

请注意我们框架支持的中间件与 `Express` 中使用的中间件的不同，这是完全正常的，也是我们如何适应这种模式以适应我们特定需求的完美演示。

使用 `ØMQ` 中间件框架

我们现在准备使用我们刚刚创建的中间件。为此，我们将构建一个非常简单的应用程序，客户端定期向服务器发送 `ping` 命令，服务器回显接收到的消息。

从实现的角度来看，我们将使用由 `ØMQ` 提供的req/rep套接字对。

然后，我们将使用我们的 `zmqMiddlewareManager` 套接字来获得我们构建的中间件，包括用于序列化/反序列化 `JSON` 消息的中间件。

服务端

首先创建服务器端（ `server.js` ）。在模块的第一部分，我们初始化我们的组件：

```
const zmq = require('zmq');
const ZmqMiddlewareManager = require('./zmqMiddlewareManager');
const jsonMiddleware = require('./jsonMiddleware');
const reply = zmq.socket('rep');
reply.bind('tcp://127.0.0.1:5000');
```

在前面的代码中，我们加载了所需的依赖关系，并将 `ØMQ rep` 套接字绑定到本地端口。接下来，我们初始化我们的中间件：

```
const zmqm = new ZmqMiddlewareManager(reply);
zmqm.use(jsonMiddleware.json());
```

我们创建了一个新的 `ZmqMiddlewareManager` 对象，然后添加了两个中间件，一个用于压缩/解压缩消息，另一个用于解析/序列化 `JSON` 消息。

> 为简洁起见，我们没有展示 `zlib` 中间件的实现，但是您可以在本书附带的示例代码中找到它。

现在我们已经准备好处理来自客户的请求。我们将通过简单地添加更多的中间件来完成这个工作，这次使用它作为请求处理程序：

```
zmqm.use({
  inbound: function(message, next) {
    console.log('Received: ', message.data);
    if (message.data.action === 'ping') {
      this.send({
        action: 'pong',
        echo: message.data.echo
      });
    }
    next();
  }
});
```

由于中间件的最后一项是在 `zlib` 和 `json` 中间件之后定义的，因此我们可以透明地使用 `message.data` 变量中可用的解压缩和反序列化消息。 另一方面，传递给 `send()` 的任何数据都将由 `outbound` 中间件处理，在我们的例子中，这个中间件将序列化，然后压缩数据。

客户端

在应用程序 `client.js` 客户端，我们首先必须启动一个连接到端口 `5000` 的新的 `ØMQ req` 套接字，这个端口是我们服务器使用的端口：

```
const zmq = require('zmq');
const ZmqMiddlewareManager = require('./zmqMiddlewareManager');
const jsonMiddleware = require('./jsonMiddleware');
const request = zmq.socket('req');
request.connect('tcp://127.0.0.1:5000');
```

然后，我们需要像我们为服务器一样设置我们的中间件框架：

```
const zmqm = new ZmqMiddlewareManager(request);
zmqm.use(jsonMiddleware.json());
```

接下来，我们创建一个中间件 `inbound` 项来处理来自服务器的响应：

```
zmqm.use({
  inbound: function(message, next) {
    console.log('Echoed back: ', message.data);
    next();
  }
});
```

在前面的代码中，我们只需拦截任何 `inbound` 响应并将其打印到控制台。

最后，我们建立一个定时器来定时发送一些 `ping` 请求，总是使用 `zmqMiddlewareManager` 来获得我们中间件的所有优点：

```
setInterval(() => {
  zmqm.send({
    action: 'ping',
    echo: Date.now()
  });
}, 1000);
```

请注意，我们正在使用 `function` 关键字明确定义所有 `inbound` 和 `outbound` 函数，避免使用箭头函数语法。这是故意的，因为正如我们在 `Chapter1-Welcome to the Node.js Platform` ，箭头函数声明将函数范围阻塞到它的词法范围。对使用箭头函数定义的函数使用调用不会改变其内部作用域。换句话说，如果我们使用箭头函数，我们的中间件将不会将其识别为 `zmqMiddlewareManager` 的一个实例，并且会引发错误 `TypeError: this.send is not a function` 。

我们现在可以通过首先启动服务器来尝试我们的应用：

```
node server
```

然后我们可以用下面的命令启动客户端：

```
node client
```

在这一点上，我们应该看到客户端发送消息和服务器回显他们。

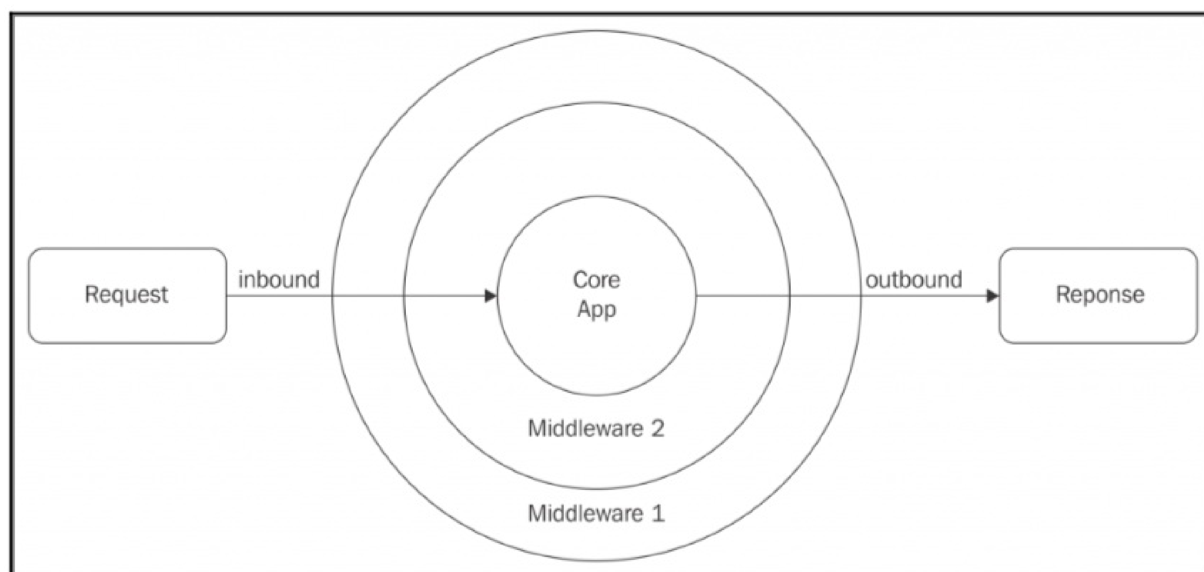
我们的中间件框架完成了它的工作。它允许我们透明地解压缩/压缩和反序列化/序列化我们的消息，让 `handler` 程序专注于他们的业务逻辑！

在 `Koa` 中使用 `Generator` 的中间件

在前面的段落中，我们看到了如何使用回调实现中间件模式，并将示例应用于消息传递系统。

正如我们在介绍它时看到的那样，中间件模式在 `Web` 框架中真正发挥作为一种便利的机制，可以构建可以在应用程序核心中处理输入和输出数据流的逻辑“层”。

除了 `Express` 之外，另一个大量使用中间件模式的 `Web` 框架是Koa。 `Koa` 是一个非常有趣的框架，主要是因为它的激进选择是只使用 `ES2015` 生成器函数而不是使用回调来实现中间件模式。我们马上就会看到这个选择如何大大简化了中间件的编写方式，但是在转移到一些代码之前，我们可以用另一种方式来形象化中间件模式，特定于这个 `Web` 框架：



在这个表示中，我们有一个传入的请求，在进入我们的应用程序的核心之前，遍历一些中间件。这部分流程称为 `inbound` 或 `downstream` 。流程到达应用程序的核心后，再遍历所有的中间件，但这次是以相反的顺序。这允许中间件在应用的主逻辑已经被执行并且响应准备好被发送给用户之后执行其他动作。 这部分流量被称为 `outbound` 或 `upstream` 。

由于中间件包装核心应用程序的方式，上面的表示有时被称为程序员的“洋葱”，这让我们想起了洋葱的层次。

现在，让我们用 `Koa` 创建一个新的 `Web` 应用程序，以了解如何使用生成器函数轻松编写定制的中间件。

我们的应用程序将是一个非常简单的 `JSON API` ，它返回我们服务器中的当前时间戳。

首先，我们需要安装 `Koa` ：

```
npm install koa
```

然后我们可以写我们的新 `app.js` ：

```
const app = require('koa')();
app.use(function*() {
  this.body = {
    "now": new Date()
  };
});
app.listen(3000);
```

需要注意的是，我们的应用程序的核心是在 `app.use` 调用中使用 `Generator` 函数定义的。我们稍后会看到中间件以完全相同的方式添加到应用程序中，并且我们将认识到，我们的应用程序的核心是最后添加到应用程序的中间件（并且不需要依赖于另一个中间件 以下项目的中间件）。

我们的应用程序的初稿已经准备就绪。 我们现在可以运行它：

```
node app.js
```

然后，我们将浏览器指向 `http://localhost:3000` ，以查看它。

请注意， `Koa` 会将响应转换为 `JSON` 字符串，并在将 `JavaScript` 对象设置为当前响应的主体时添加正确的内容类型标头。

我们的 `API` 运行良好，但是现在我们可能会决定保护它免受滥用，确保人们在一秒钟内完成多个请求。 这个逻辑可以被认为是我们 `API` 的业务逻辑的外部，所以我们应该通过简单地写一个新的专用中间件来添加它。我们把它写成一个叫做 `rateLimit.js` 的独立模块：

```
const lastCall = new Map();

module.exports = function *(next) {

  // inbound
  const now = new Date();
  if (lastCall.has(this.ip) && now.getTime() - lastCall.get(this
.ip).getTime() < 1000) {
    return this.status = 429; // Too Many Requests
  }

  yield next;

  // outbound
  lastCall.set(this.ip, now);
  this.set('X-RateLimit-Reset', now.getTime() + 1000);
};
```

我们的模块导出一个实现我们中间件逻辑的生成器函数。

首先要注意的是，我们使用 `Map` 对象来存储从给定 `IP` 地址接收到最后一次呼叫的时间。我们将使用这个 `Map` 作为一种内存数据库，能够检查一个特定的用户是否每秒钟以超过一个请求来超载我们的服务器。当然，这个实现仅仅是一个虚拟的例子，在真实的情况下这并不理想，只使用外部存储（如 `Redis` 或 `Memcache` ）和更精确的逻辑来检测过载。

我们可以看到，中间件的主体被分成两个逻辑部分， `inbound` 和 `outbound` ，与下一个 `yield` 的分离。在 `inbound` 部分，我们还没有走到应用程序的核心，所以这是我们需要检查用户是否超出我们的费率限制的地方。如果是这样，我们只需将响应的 `HTTP` 状态码设置为 `429` （ `too many requests` ），我们返回来停止 `pipeline` 的执行。

另一个我们可以进入下一个中间件的方法是通过 `next` 调用 `yield` 。使用 `Generator` 函数和 `yield` ，中间件的执行被暂停，以执行列表中的所有其他中间件，并且只有当中间件的最后一项被执行时（应用程序的真正核心） `outbound` 流程可以开始，并且以相反的顺序将控制权交还给每个中间件，直到第一个中间件再次被调用。

当我们的中间件再次接收到控制信号并且恢复 `Generator` 功能时，我们需要保存成功调用的时间戳，并且在请求中添加一个 `X-RateLimit-Reset` 头，以表示用户何时能够创建一个新的请求。

> 如果你需要一个更完整和可靠的限速中间件的实现，你可以看看 `koajs/ratelimit` 模块，https://github.com/koajs/ratelimit

为了启用这个中间件，我们需要在包含我们应用的核心逻辑的现有 `app.use` 之前在我们的 `app.js` 中添加以下行：

```
app.use(require('./rateLimit'));
```

现在看到我们的新应用程序在运行，我们需要重新启动我们的服务器，再次打开我们的浏览器。如果我们快速刷新页面几次，我们可能会达到速率限制，我们应该看到描述错误消息“太多请求”。由于将状态码设置为 `429` 并具有空的响应主体， `Koa` 自动添加此消息。

> 如果您有兴趣阅读基于 `Koa` 框架中使用的生成器的中间件模式的实际实现，您可以查看koajs/compose，它是核心模块用于将一组 `Generator` 转换成一个新的 `Generator` ，该 `Generator` 在 `pipeline` 中执行原始 `Generator` 。

```
~/workspace
➜  workspace curl localhost:3000
{"now":"2018-01-29T12:44:40.393Z"}%
➜  workspace
```

总结

在本章中，我们了解了如何将一些传统的 `GOF` 设计模式应用于 `JavaScript` ，特别是 `node.js` 。其中一些被转换，一些被简化，另一些被重新命名或被改编，作为它们被语言、平台和社区同化的一部分。我们强调了简单的模式(如工厂模式)如何极大地提高代码的灵活性，以及如何使用代理、装饰器和适配器来操作、扩展和调整现有对象的接口。相反，策略模式、状态模式和模板模式已经向我们展示了如何将更大的算法分解为静态和可变的部分，从而使我们能够提高组件的代码重用性和可扩展性。通过学习中间件模式，我们现在能够使用简单、可扩展和优雅的范例来处理数据。最后，命令模式为我们提供了一个简单的抽象，使任何操作都更加灵活和强大。

除了观察这些被广泛接受的设计模式的 `JavaScript` 版本，我们还发现了一些在 `JavaScript` 社区中诞生和提出的新的设计模式，例如揭示构造函数和可组合的工厂函数模式。这些模式有助于处理 `JavaScript` 语言的特定方面，例如 `asynchronicity` 和 `prototype-based programming` 。

最后，我们获得了更多的证据，说明 `JavaScript` 是如何通过组合不同的可重用对象或函数来完成任务和构建软件的，而不是扩展许多小类或接口。此外，对于来自其他面向对象语言的开发人员来说，看到一些设计模式在 `JavaScript` 中实现时有多么不同可能会显得很奇怪；有些人可能会感到迷茫，因为知道可能不止一种设计模式，而是许多实现设计模式的不同方式。我们说， `JavaScript` 是一种实用的语言，它允许我们快速完成任务，但是，没有任何结构或指导原则，我们就会自找麻烦。这就是这本书，尤其是这一章有用的地方。它试图在创造力和严谨性之间教出正确的平衡。它不仅显示了可以重用的模式来改进我们的代码，而且它们的实现不是最重要的细节；它可能与其他模式有很大的不同，甚至重叠。真正重要的是蓝图、指导方针和模式基础上的想法。这是真正可重用的信息，我们可以利用这些信息以有趣的方式设计更好的 `node.js` 应用程序。

在下一章中，我们将分析更多的设计模式，重点是编程的一个最有主见的方面：如何将模块组织起来并连接在一起。

Writing Modules

`Node.js` 模块系统弥补了原生 `JavaScript` 缺乏把代码组织到不同独立单元的这一缺陷。模块系统最大的优点就是能够使用 `require()` 函数将模块链接在一起，这是一种简单而强大的方法。但是，对于许多新的 `Node.js` 的开发人员可能会对模块系统的使用产生疑问。实际上，最常见的问题之一是：将组件X的实例传递到模块Y的最佳方式是什么？

有时候，这种疑问可能导致我们滥用单例模式，因为希望找到一种更熟悉的方式来将我们的模块链接在一起。另一方面，我们可能滥用依赖注入模式，利用它来处理任何类型的依赖（甚至无状态）。如果说如何组织模块是 `Node.js` 中最具争议性和观点性的话题之一应该不足为奇了。主流的组织模块方式很多，但没有任意一个观点处于主导地位。但实际上，每种方法都有其优点和缺点。

在本章中，我们将分析组织模块的各种方法，并强调它们的优缺点，以便我们能够在简单性，可重用性和可扩展性之间平衡，合理地选择和混用这些模块组织方式。具体来说，我们将介绍一些模式，如下所示：

- 硬编码依赖
- 依赖注入
- 服务定位器
- 依赖注入容器

然后，我们将探讨一个与书写模块密切相关的问题，即如何组织 `Node.js` 插件模块。对于这个问题，大多数书写插件模块的方式都差不多，但是与用户自己编写的应用程序模块的组织就不太相同了，特别是当插件作为单独的 `Node.js` 包分发时，问题就十分明显了。

我们将学习如何构建一个 `Node.js` 插件，并如何把这些插件集成到主应用程序中。

在本章最后，对于 `Node.js` 如何组织模块就不再是晦涩难懂的话题了。

模块和依赖

每个应用程序都是多个模块组织在一起的结果，如同盖楼一样，随着应用程序日益迭代复杂，我们组织模块的方式将导致应用程序的成功或失败。这不仅与应用程序的拓展性相关，还是我们构建大型系统的重点关注点。过于复杂紊乱的模块依赖是一种灾难，它增加了我们项目的组织难度，在这种情况下，代码的任何修改和拓展都将会使我们付出巨大的代价。

最糟糕的情况是，这些模块严重耦合，导致我们不重写整个应用程序就不更改代码的任何一部分。当然，不必害怕，我们并不用从写第一个模块开始就开始全面规划我们的模块。但只要我们遵循应有的模式，就不会出现这样的问题。

`Node.js` 提供了一个很好的工具来连接和组织应用程序。那就是 `CommonJS` 模块系统。但是，使用模块系统并不能够保证我们我们一定能解决模块依赖的问题，如果使用不当，将会使得耦合变得更加严重。在本节中，我们将讨论书写 `Node.js` 模块的基本模式。

Node.js最常见的依赖

在一个软件体系结构中，我们在设计其的过程中就应该考虑到可能影响其中任何一个组件依赖关系的实体、状态、数据格式。例如，一个组件可能使用另一个组件的提供的服务，也可能依赖系统特定的一个全局状态，或者实现一个特定的通信协议，以便与其他组件交换信息等等。依赖的概念十分广泛，有时会显得难以评估。

但是，在 `Node.js` 中，我们可以确定一个最常见也最容易识别的最基本的依赖模型。当然，当我们在讨论模块之间的依赖关系，我们应该首先明确：模块是我们组织和构建代码的基本机制。不依赖模块系统构建的大型应用程序是十分不合理的。如果使用正确的方式来组织应用程序的各个模块单元，它会带来很多好处。实际上，一个模块的属性可以概括如下：

- 一个模块应该具有可读性和可理解性，因为它应该专注于一件事
- 一个模块被表示为一个单独的文件，使得其更容易被识别
- 模块可以更容易地在不同的应用程序中复用

一个模块代表的是一个完全私有的命名空间，并通过 `module.exports` 来公开访问这个模块的接口。

但是，对于一个成功的模块设计，只是简单地将应用程序或库的功能区分为不同的模块是完全不够的。最常见的错误会出现在我们创建了一个过于复杂的模块，那么想要替换或更改这个模块会对整个应用的架构产生巨大的影响。这时就能够意识到把代码组织成模块的优势了。我们需要在模块设计中找到一个平衡点。

内聚与耦合

评判创建的模块平衡性两个最重要的特征就是内聚度和耦合度。这两个特征可以应用于软件体系结构中的任何类型的组件或子系统。因此在构建 `Node.js` 模块时也可以把这两个特征作为重要的参考价值。这两个属性定义如下：

- 内聚度：用于度量模块内部功能之间的相关性。例如，对于一个只做一件事的模块，其中的所有部件都只对这一件事起作用，那说明这个模块具有很高的内聚度。举个例子，那种包含把任何类型的对象存储到数据库的函数内聚度就较低，如 `saveProduct()` 、 `saveInvoice()` 、 `saveUser()` 等。

- 耦合度：评判模块对系统其他模块的依赖程度。例如，当一个模块直接读取或修改另一个模块的数据时，该模块与另一个模块紧密耦合。另外，通过全局或共享状态交互的两个模块紧密耦合。另一方面，仅通过参数传递进行通信的两个模块耦合度较低。

理想情况下，一个模块应该具有较高的内聚度和较低的耦合度，这样的模块更易于理解、重用和扩展。

有状态模块

在 `JavaScript` 中，一切都是对象。它没有纯粹的类或者接口的概念，因为其动态类型的机制，已经将接口或者策略和实现细节分开。这就是为什么我们在 `Chapter 6-Design Patterns` 看到在 `JavaScript` 中一些设计模式和传统的设计模式看起来如此不同并且简单的多的原因。

在JavaScript中，将接口与实现分离的例子很少。 然而，通过使用 `Node.js` 模块系统，我们引入了一个特定的模块，接口不会受到其它模块的影响。在正常情况下，这没有什么问题，但是如果我们使用 `require()` 来加载一个导出有状态实例的模块，比如数据库交互对象，HTTP服务器实例，乃至普通的任何对象这不是无状态的，我们实际上是在引用的模块都是一个又一个的单例，因此模块系统有着单例模式的优点和缺点，此外，也有一些不同的地方。

Node.js的单例模式

很多刚接触 `Node.js` 的人对于如何正确地实现单例模式感到困惑，通常情况下，应用程序的各个模块之间共享一个实例。 `Node.js` 中要想实现这一点特别简单；只需使用 `module.exports` 导出实例就足以获得与 `Singleton` 模式非常相似的效果。

例如，考虑下面这行代码：

```
//'db.js' module
module.exports = new Database('my-app-db');
```

通过导出 `Database` 的一个实例，我们可以假定在当前包（这可以很容易地成为我们应用程序的整个代码）内，我们将只有一个 `db` 模块的实例。这是可能的，因为我们知道， `Node.js` 将在第一次调用 `require()` 之后缓存模块，确保在随后的调用中不再执行它，而是返回缓存实例。例如，我们可以很容易地获得我们之前定义的 `db` 模块的一个共享实例，使用下面这行代码：

```
const db = require('./db');
```

但是注意，该模块使用的是相对路径引入，因此其是符合单例模式的。我们在 `Chapter2-Node.js Essential Patterns` 中看到，每个包在其 `node_modules` 目录中都可能有自己的一组专用依赖项，这可能会导致同一个模块会有多个实例，例如，考虑将 `db` 模块封装到名为 `mydb` 的包中的情况。看以下代码 `package.json` 文件中的代码：

```
{
  "name": "mydb",
  "main": "db.js"
}
```

现在考虑下面的依赖包的关系树：

```
app/
   `-- node_modules
       |-- packageA
       |  `-- node_modules
       |      `-- mydb
       `-- packageB
           `-- node_modules
               `-- mydb
```

`packageA` 和 `packageB` 都依赖于 `mydb` 模块；反过来，其它的应用程序模块，可能同时依赖于 `packageA` 和 `packageB` 。我们刚刚描述的场景将打破关于数据库实例唯一性的假设；实际上， `packageA` 和 `packageB` 都将使用如下命令加载 `db` 实例：

```
const db = require('mydb');
```

然而， `packageA` 和 `packageB` 实际上会加载两个不同的单例，因为 `mydb` 模块将根据所需的包来解析到不同的目录。

在这一点上，我们可以很容易地说，除非我们使用真正的全局变量来存储一个模块实例，否则之前描述的单例模式在 `Node.js` 中不存在，如下所示：

```
global.db = new Database('my-app-db');
```

这将保证该实例将是唯一的，并在整个应用程序中共享，仅仅是在一个模块中。但是，我们应该尽量避免这么做。在大多数情况下，我们并不需要一个纯粹的单例模式，无论如何，我们稍后会看到，还有其他模式可以用来在不同的包中共享一个实例。

> 在本书中，为了简单起见，我们将使用术语单例模式来描述由模块导出的有状态对象，即使这并不代表严格定义的单一实例。但是，我们可以肯定地说，它与原始的单例模式具有相同的含义：可以在不同的组件之间共享状态。

书写模块的模式

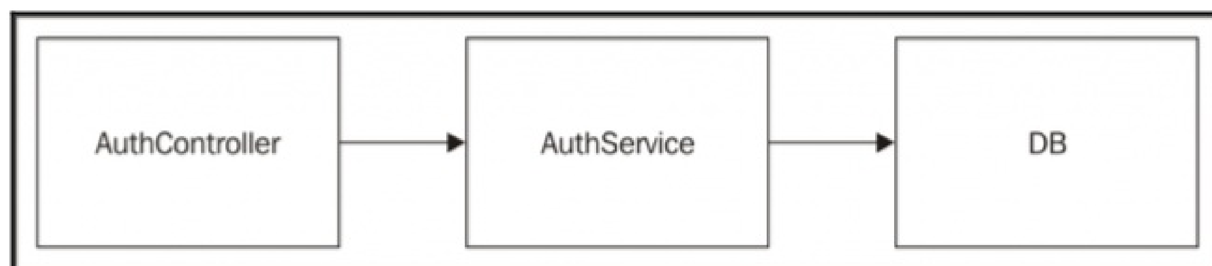
现在我们已经讨论了一些关于内聚和耦合的基本理论，我们已经准备好了一些更实际的概念。实际上，在这一节中，我们将介绍怎么书写模块。我们重点讲解如何利用有状态模块实例，毫无疑问，它是应用程序中最重要的一类依赖。

硬编码依赖

我们开始通过分析两个模块之间最常见的关系来看硬编码依赖。在 `Node.js` 中，当一个客户端模块使用 `require()` 加载另一个模块时就会建立模块的硬编码依赖关系。正如我们将在本节中看到的，这种建立模块依赖关系的方法简单而有效，但是我们必须更加关注有状态实例的硬编码依赖关系，否则在有状态实例模块会限制我们的模块复用。

使用硬编码的依赖关系构建鉴权服务

我们从下图所示的结构开始分析：



上图显示了分层体系结构的典型示例；它描述了一个简单的鉴权服务的结构。 `AuthController` 接受来自客户端的输入，从请求中提取登录信息，并执行一些初步验证。之后 `AuthService` 检查客户端提供的凭证是否与存储在数据库中的信息匹配；这是通过使用 `db` 模块执行一些特定的查询来完成的，作为与数据库通信的一种手段。这三个组件连接在一起的方式将决定它们的可重用性，可测试性和可维护性的强度。

将这些组件连接在一起的最自然的方法是通过 `AuthService` 请求 `db` 模块，然后从 `AuthController` 请求 `AuthService` 。 这是我们正在讨论的硬编码依赖。

让我们通过实际实现刚刚描述的系统来演示这一点。那么我们来设计一个简单的鉴权服务器，它将有以下两个 `HTTP API` ：

- `POST '/ login'` ：接收包含用户名和密码对进行身份验证的 `JSON` 对象。成功时，它会返回一个 `JSON Web Token（JWT）` ，随后的请求中使用它来验证用户的身份。

> `JSON Web Token` 是一种客户端和服务端身份验证的格式。但随着单页应用程序和跨源资源共享（CORS）技术的增长，基于 `cookie` 的身份验证的更为灵活的替代方案，其受欢迎程度正在不断提高。要了解更多关于 `JSON Web Token` 的信息，可以参考http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html上的规范

- `GET'/ checkToken'` ：查看用户是否具有权限。

对于这个例子，我们将使用几种技术；其中一些对我们来说并不陌生。我们使用express来实现 `Web API` 和levelup来存储用户的数据。

db模块

我们先从底层开始构建应用程序；我们需要的第一件事就是公开一个 `levelUp` 数据库实例的模块。 我们来创建一个名为 `lib/db.js` 的新文件，其中包含以下内容：

```
const level = require('level');
const sublevel = require('level-sublevel');
module.exports = sublevel(
  level('example-db', {
    valueEncoding: 'json'
  })
);
```

前面的模块只是创建一个到存储在 `./example-db` 目录中的 `LevelDB` 数据库的连接，然后使用sublevel来修饰实例，该插件添加了支持增删查改数据库（可以将其与 `SQL` 或 `MongoDB` 进行比较）。模块导出的对象是数据库对象本身，它是一个有状态的实例；因此，我们创建的是单例。

authService模块

现在我们有了 `db` 单例，我们可以使用它来实现 `lib/authService.js` 模块，它负责查询数据库，根据用户身份凭证查看用户是否具有权限。 代码如下（只显示相关部分）：

```
"use strict";

const jwt = require('jwt-simple');
const bcrypt = require('bcrypt');

const db = require('./db');
const users = db.sublevel('users');

const tokenSecret = 'SHHH!';

exports.login = (username, password, callback) => {
  users.get(username, (err, user) => {
    if(err) return callback(err);

    bcrypt.compare(password, user.hash, (err, res) => {
      if(err) return callback(err);
      if(!res) return callback(new Error('Invalid password'));

      let token = jwt.encode({
        username: username,
        expire: Date.now() + (1000 * 60 * 60) //1 hour
      }, tokenSecret);

      callback(null, token);
    });
  });
};

exports.checkToken = (token, callback) => {
  let userData;
  try {
    //jwt.decode will throw if the token is invalid
    userData = jwt.decode(token, tokenSecret);
    if (userData.expire <= Date.now()) {
      throw new Error('Token expired');
    }
  } catch(err) {
    return process.nextTick(callback.bind(null, err));
  }

  users.get(userData.username, (err, user) => {
    if (err) return callback(err);
    callback(null, {username: userData.username});
  });
};
```

`authService` 模块实现 `login()` 服务，该服务负责查询数据库，检查用户名和密码信息， `checkToken()` 服务接受 `token` 作为参数并验证其有效性。

上面的代码是有状态模块的硬编码依赖关系的第一个示例。我们正在谈论 `db` 模块，我们只需要加载它。生成的 `db` 变量包含一个已经初始化的数据库对象，我们可以直接使用它来执行我们的查询。

在这一点上，我们可以看到，我们为 `authService` 模块创建的所有代码并不需要 `db` 模块的一个特定实例，任何实例都可以正常发挥作用。但是， `authService` 模块硬编码依赖于 `levelUp` 数据库对象实例，这意味着我们将无法在不更改其模块本身代码的情况下将 `authService` 与另一个数据库实例结合使用。

authController模块

继续在应用程序的层次上，我们现在要看看 `lib/authController.js` 模块。这个模块负责处理 `HTTP` 请求，它本质上是 `Express` 路由的集合；该模块的代码如下：

```
"use strict";

const authService = require('./authService');

exports.login = (req, res, next) => {
  authService.login(req.body.username, req.body.password,
    (err, result) => {
      if (err) {
        return res.status(401).send({
          ok: false,
          error: 'Invalid username/password'
        });
      }
      res.status(200).send({ok: true, token: result});
    }
  );
};

exports.checkToken = (req, res, next) => {
  authService.checkToken(req.query.token,
    (err, result) => {
      if (err) {
        return res.status(401).send({
          ok: false,
          error: 'Token is invalid or expired'
        });
      }
      res.status(200).send({ok: 'true', user: result});
    }
  );
};
```

`authController` 模块实现两个 `Express` 路由： `login()` 用于执行登录操作并返回相应的 `token` ， `checkToken()` 用于检查 `token` 的有效性。这两个路由委托他们的大部分逻辑到 `authService` ，所以他们唯一的工作是处理 `HTTP` 请求和响应。

我们也可以看到，在这种情况下，我们使用有状态模块 `authService` 来硬编码依赖项。是的， `authService` 模块通过传递性是有状态的，因为它直接依赖于 `db` 模块。 有了这个，我们理解了硬编码的依赖关系如何贯穿整个应用程序的结构中： `authController` 模块依赖于 `authService` 模块，而 `authService` 模块依赖于 `db` 模块；这意味着 `authService` 模块本身是间接链接到一个特定的数据库实例的。

app模块

最后，在应用程序的入口点，我们调用我们的 `controller` 。遵循约定，我们将把这个逻辑放在名为 `app.js` 的模块中，放在我们项目的根目录下，如下所示：

```
"use strict";

const Express = require('express');
const bodyParser = require('body-parser');
const errorHandler = require('errorhandler');
const http = require('http');

const authController = require('./lib/authController');

let app = module.exports = new Express();
app.use(bodyParser.json());

app.post('/login', authController.login);
app.get('/checkToken', authController.checkToken);

app.use(errorHandler());
http.createServer(app).listen(3000, () => {
  console.log('Express server started');
});
```

我们可以看到，我们的应用程序模块是非常基础的。 它包含一个简单的 `Express` 服务器，它注册了一些中间件和 `authController` 导出的两条路由。当然，对于我们来说最重要的代码是 `authController` 所导出的硬编码依赖实例。

运行鉴权服务

在我们尝试我们刚刚实现的认证服务器之前，我们建议您使用代码示例中提供的 `populate_db.js` 脚本来填充数据库中的一些示例数据。 这样做之后，我们可以通过运行以下命令来启动服务器：

```
node app
```

然后我们可以尝试调用我们创建的两个 `Web` 服务; 我们可以使用 `REST` 客户端来执行此操作，或者使用旧的 `curl` 命令。 例如，要执行登录，我们可以运行以下命令：

```
curl -X POST -d '{"username": "alice", "password":"secret"}' http://localhost:3000/login -H "Content-Type: application/json"
```

前面的命令应该返回一个 `token` ，我们可以使用它来测试 `/checkLogin` 的Web服务（只需输入以下命令并替换 `<TOKEN HERE>` ）：

```
curl -X GET -H "Accept: application/json" http://localhost:3000/checkToken?token=<TOKEN HERE>
```

前面的命令应该返回一个字符串，如下所示，这确认我们的服务器正在按预期工作：

```
{"ok":"true","user":{"username":"alice"}}
```

硬编码依赖的优点和缺点

我们刚刚实现的示例演示了 `Node.js` 中书写模块的传统方式以及利用模块系统的全部功能来管理应用程序各个组件之间的依赖关系。我们从模块中导出有状态的实例，让 `Node.js` 管理它们的生命周期，然后我们直接从应用程序的其他部分引入它们。这样管理起来非常直观，易于理解和调试，每个模块初始化和引入，都不会受到任何外部条件的干预。

然而，另一方面，对有状态实例的依赖性进行硬编码会限制将模块与其他实例关联的可能性，这使得在单元测试的过程中，其可重用性更低，测试难度更大。例如，将 `authService` 与其他数据库实例结合使用几乎是不可能的，因为它的依赖关系是用一个特定的实例进行硬编码的。同样，单独测试 `authService` 可能是一件困难的事情，因为我们不能轻易地模拟另一模块使用数据库。

最后，重要的是要看到使用硬编码依赖的大多数缺点都与有状态的实例相关联。这意味着如果我们使用 `require()` 来加载一个无状态模块，例如一个工厂，构造函数或者一组无状态函数，我们就不会遇到同样的问题。我们仍然会与特定的实现紧密耦合，但在 `Node.js` 中，这通常不会影响组件的可重用性，因为在模块内部创建的实例不会引入与特定状态的耦合。

依赖注入

依赖注入（DI）模式可能是软件设计中最容易被误解的概念之一。许多人将这个术语与框架和依赖注入容器相关联，例如 `Spring` （用于 `Java` 和 `C#` ）或 `Pimple` （用于 `PHP` ），但实际上它是一个很简单的概念。依赖注入模式背后的主要思想是由外部实体提供输入的组件的依赖关系。

这样的实体可以是客户端组件或全局容器，它集中了系统所有模块的关联。这种方法的主要优点是解耦，特别是对于取决于有状态实例的模块。使用DI，从外部接收每个依赖项，而不是硬编码到模块中。这意味着模块可以配置为其中的依赖关系，因此可以在不同的上下文中重用。

为了在实践中演示这种模式，我们现在要重构我们在前一节中构建的鉴权服务器，使用DI来连接它的模块。

使用DI重构鉴权服务器

使用DI重构我们的模块是很简单的：我们不需要将依赖关系硬编码到有状态实例，而是创建一个工厂，它将一组依赖作为参数。

让我们立即开始这个重构; 让我们来看看如下的 `lib/db.js` 模块：

```
"use strict";

const level = require('level');
const sublevel = require('level-sublevel');

module.exports = function(dbName) {
  return sublevel(
    level(dbName, {valueEncoding: 'json'})
  );
};
```

重构过程的第一步是将 `db` 模块转换为工厂模式。结果是我们现在可以使用它创建尽可能多的数据库实例，这意味着整个模块现在可以重用和无状态。

我们继续并实现新版本的 `lib/authService.js` 模块：

```
"use strict";

const jwt = require('jwt-simple');
const bcrypt = require('bcrypt');

module.exports = (db, tokenSecret) => {
  const users = db.sublevel('users');
  const authService = {};

  authService.login = (username, password, callback) => {
    users.get(username, (err, user) => {
      if(err) return callback(err);

      bcrypt.compare(password, user.hash, (err, res) => {
        if(err) return callback(err);
        if(!res) return callback(new Error('Invalid password'));

        const token = jwt.encode({
          username: username,
          expire: Date.now() + (1000 * 60 * 60) //1 hour
        }, tokenSecret);

        callback(null, token);
      });
    });
  };

  authService.checkToken = (token, callback) => {
    let userData;
    try {
      //jwt.decode will throw if the token is invalid
      userData = jwt.decode(token, tokenSecret);
      if (userData.expire <= Date.now()) {
        throw new Error('Token expired');
      }
    } catch(err) {
      return process.nextTick(callback.bind(null, err));
    }

    users.get(userData.username, (err, user) => {
      if(err) return callback(err);
      callback(null, {username: userData.username});
    });
  };

  return authService;
};
```

此外， `authService` 模块现在是无状态的; 它不再导出任何特定的实例，只是一个简单的工厂。 但最重要的细节是，我们将 `db` 依赖注入作为工厂函数的一个参数，删除以前的硬编码码依赖。这个简单的更改使我们能够通过将它连接到任何数据

库实例来创建一个新的 `authService` 模块。

我们可以用类似的方式重构 `lib/authController.js` 模块，如下所示：

```
"use strict";

module.exports = (authService) => {
  const authController = {};

  authController.login = (req, res, next) => {
    authService.login(req.body.username, req.body.password,
      (err, result) => {
        if (err) {
          return res.status(401).send({
            ok: false,
            error: 'Invalid username/password'
          });
        }
        res.status(200).send({ok: true, token: result});
      }
    );
  };

  authController.checkToken = (req, res, next) => {
    authService.checkToken(req.query.token,
      (err, result) => {
        if (err) {
          return res.status(401).send({
            ok: false,
            error: 'Token is invalid or expired'
          });
        }
        res.status(200).send({ok: 'true', user: result});
      }
    );
  };

  return authController;
};
```

`authController` 模块根本没有任何硬编码依赖，甚至没有状态。唯一的依赖 `authService` 模块在调用时作为输入提供给工厂。

好吧，现在是时候看看所有这些模块是在哪里创建和连接在一起的。答案在于 `app.js` 模块，它代表了我们应用程序中的最顶层。其代码如下：

```
"use strict";

const Express = require('express');
const bodyParser = require('body-parser');
const errorHandler = require('errorhandler');
const http = require('http');

const app = module.exports = new Express();
app.use(bodyParser.json());

const dbFactory = require('./lib/db');
const authServiceFactory = require('./lib/authService');
const authControllerFactory = require('./lib/authController');

const db = dbFactory('example-db');
const authService = authServiceFactory(db, 'SHHH!');
const authController = authControllerFactory(authService);

app.post('/login', authController.login);
app.get('/checkToken', authController.checkToken);

app.use(errorHandler());
http.createServer(app).listen(3000, () => {
  console.log('Express server started');
});
```

前面的代码可以概括如下：

1. 我们加载 `services` 的工厂；在这一点上，其仍然是无状态的对象。
2. 我们通过引入它所需的依赖来实例化每个服务。这是模块创建和链接的阶段。
3. 最后，我们像往常一样在 `Express` 服务器上注册 `authController` 模块的路由。

鉴权服务器现在使用 `DI` 链接，提高了其复用性。

DI的不同类型

我们刚刚介绍的例子只演示了一种类型的 `DI` （工厂注入），但是还有一些类型的 `DI` 更值得一提：

- 构造函数注入：在这种类型的 `DI` 中，依赖关系在创建时传递给构造函数；一个可能的例子可以是：

```
const service = new Service(dependencyA, dependencyB);
```

- 属性注入：在这种类型的 `DI` 中，依赖关系在创建之后附加到对象上，如以下代码所示：

```
const service = new Service();
service.dependencyA = anInstanceOfDependencyA;
```

属性注入意味着一个对象会被创建为不一致的状态，因为它没有连接到它的依赖关系，所以它是最不健壮的，但是当依赖关系之间存在循环时，它有时可能是有用的。例如，如果我们有两个组件A和B，它们都使用工厂或构造函数注入，并且都相互依赖，我们不能实例化它们中的任何一个，因为两者都需要另一个存在才能被创建。我们来看一个简单的例子，如下所示：

```
function Afactory(b) {
  return {
    foo: function() {
      b.say();
    },
    what: function() {
      return 'Hello!';
    }
  }
}

function Bfactory(a) {
  return {
    a: a,
    say: function() {
      console.log('I say: ' + a.what);
    }
  }
}
```

前两个工厂之间的依赖关系死锁只能通过属性注入来解决，例如先创建一个不完整的 `B` 实例，然后才能创建 `A` 。最后，我们将 `A` 注入到 `B` 中，方法是设置相关属性 如下：

```
const b = Bfactory(null);
const a = Afactory(b);
a.b = b;
```

> 在极少数情况下，依赖图中的循环是不容易避免的; 然而，重要的是要记住，这往往是一个糟糕的设计，应该尽可能避免。

DI的优点和缺点

在使用 `DI` 的鉴权服务器示例中，我们能够将我们的模块与特定的依赖项实例分离。结果是，我们现在可以用最少的代价复用每个模块，而且代码没有任何改变。测试使用 `DI` 模式的模块也大大简化；我们可以轻松地模拟模块的依赖关系，并且独立于系统其他部分的状态来测试我们的模块。

我们前面介绍的例子中要强调的另一个重要方面是，我们将依赖链接的地方从底层移到了顶层。

这个想法是，高级组件在本质上比低级组件更不易重复使用，这是因为我们在应用程序的层次越多，组件越具体。

基于这个假设，那么高级组件底层依赖关系的应用程序架构的顺序是可以颠倒的，这样底层组件只依赖于一个接口（在 `JavaScript` 中，它是只是我们期望的一个依赖的接口），而定义一个依赖的实现的所有权是给予更高级别的组件的。在我们的鉴权服务器中，实际上，所有的依赖关系都被实例化，并被连接到最上面的组件，即我们的应用程序模块（ `app.js` ），这也是不太可重用的，并且耦合度较高。

所以耦合度和复用性是相悖的。通常，如果编码时无法解决依赖关系，理解系统各个组件之间的关系就会变得更困难。另外，如果我们看一下我们在应用程序模块中实例化所有依赖的方式，我们可以看到我们必须遵循特定的顺序。我们实际上不得不手动构建整个应用程序的依赖关系图。当要链接的模块数量变多时，这可能变得难以管理。

解决这个问题的一个可行的解决方案是在多个组件之间拆分依赖，而不是集中在一个地方。这可以减少涉及管理依赖关系的复杂度，因为每个组件只负责其特定的依赖关系子图。当然，我们也可以选择仅在本地使用 `DI` ，只是在必要时使用 `DI` ，而不是在整个应用程序之上构建。

我们将在本章后面看到，另一种简化复杂体系结构中模块连接的可能解决方案是使用一个DI容器，一个专门负责实例化和连接应用程序所有依赖关系的组件。

使用 `DI` 肯定会增加我们模块的复杂性和冗长度，但正如我们前面所看到的，这样做有很多好的理由。取决于我们想要获得的简单性和可重用性之间的平衡，至于选择依赖注入还是选择硬编码依赖，则取决于我们。

> DI经常结合 `Dependency Inversion principle（依赖倒置准则）` 和 `Inversion of Control（控制反转）` 一并讨论; 然而，他们虽然相关，但却是不同的概念。

服务定位器

在前面的章节中，我们学习了 `DI` 如何通过获得可重用和解耦的模块连接依赖关系。与这一模式相类似的另一种模式是服务定位器。服务定位器核心原则是拥有一个中央注册中心，以便管理系统组件，并在模块需要加载依赖时作为中介。这个想法是要求服务定位器所连接的是依赖注入模块，而不是硬编码模块。

理解这一点很重要，通过使用服务定位器，我们引入了对它的依赖关系，它连接到模块的方式决定了它们的耦合程度，其可重用性较高。 在 `Node.js` 中，我们可以确定三种类型的服务定位器，区分它们的关键因素是它们连接到系统各个组件的方式：

- 硬编码依赖服务定位器
- 依赖注入服务定位器
- 全局注入服务定位器

硬编码依赖服务定位器耦合度较高，因为它由使用 `require()` 直接引入服务定位器的实例组成。在 `Node.js` 中，这可以被认为是一种反模式，因为它引入了一个紧密耦合的组件。在这种情况下，服务定位器在重用性方面显然没有提供任何价值，只是增加了另一层级的间接性和复杂性。因此应该抛弃硬编码依赖服务定位器这种模块引入方式。

依赖注入服务定位器通过 `DI` 引用组件。这可以被认为是一次注入一整套依赖的更方便的方法，而不是一个接一个地提供它们。而且我们将看到它的优势并不止于此。

全局注入服务定位器直接注入到全局。这与硬编码服务定位器具有相同的缺点，但由于它是全局的，因此它是一个真正的单例，因此可以很容易地用作包之间共享实例的模式。我们将在后面的章节中看到这一点，但现在我们可以肯定地说，全局注入服务定位器使用场景更少。

> `Node.js` 模块系统已经实现了服务定位器模式的变体，其中 `require()` 代表服务定位器本身的全局实例。

一旦我们开始使用服务定位器模式，上述所说的将变得更加清晰。现在重构鉴权服务器来实践服务定位器。

使用服务定位器重构鉴权服务

我们现在要使用服务定位器重构鉴权服务器。要做到这一点，第一步是实现服务定位器本身；我们将使用一个新的模块 `lib/serviceLocator.js` ：

```
"use strict";

module.exports = () => {
  const dependencies = {};
  const factories = {};
  const serviceLocator = {};

  serviceLocator.factory = (name, factory) => {
    factories[name] = factory;
  };

  serviceLocator.register = (name, instance) => {
    dependencies[name] = instance;
  };

  serviceLocator.get = (name) => {
    if (!dependencies[name]) {
      const factory = factories[name];
      dependencies[name] = factory && factory(serviceLocator);
      if (!dependencies[name]) {
        throw new Error('Cannot find module: ' + name);
      }
    }
    return dependencies[name];
  };

  return serviceLocator;
};
```

我们的 `serviceLocator` 模块是一个用三种方法返回对象的工厂函数：

- `factory()` 方法用于将组件名称与工厂函数关联。
- `register()` 用于将组件名称直接与实例相关联。
- `get()` 通过名称检索组件。如果一个实例已经可用，它只是返回它；否则，它会尝试调用注册的工厂来获取新的实例。注意到模块工厂是通过注入服务定位器（ `serviceLocator` ）的当前实例来调用是非常重要的。这是模式的核心机制，允许自动和按需建立系统依赖关系图。接下来看它是如何工作的。

> 服务定位器使用一个对象作为一组依赖项的命名空间：

```
const dependencies = {};
const db = require('./lib/db');
const authService = require('./lib/authService');
dependencies.db = db();
dependencies.authService = authService(dependencies);
```

更改 `lib/db.js` 模块来 `serviceLocator` 的工作：

```
"use strict";

const level = require('level');
const sublevel = require('level-sublevel');

module.exports = (serviceLocator) => {
  const dbName = serviceLocator.get('dbName');

  return sublevel(
    level(dbName, {valueEncoding: 'json'})
  );
};
```

`db` 模块使用输入中接收到的服务定位器来检索要实例化的数据库的名称。需要强调的是，服务定位器不仅可用于返回组件实例，还可用于提供定义我们要创建的整个依赖关系图的行为的配置参数。

接下来更改 `lib/authService.js` 模块：

```
"use strict";

const jwt = require('jwt-simple');
const bcrypt = require('bcrypt');

module.exports = (serviceLocator) => {
  const db = serviceLocator.get('db');
  const tokenSecret = serviceLocator.get('tokenSecret');

  const users = db.sublevel('users');
  const authService = {};

  authService.login = (username, password, callback) => {
    users.get(username, (err, user) => {
      if (err) return callback(err);

      bcrypt.compare(password, user.hash, (err, res) => {
        if (err) return callback(err);
        if (!res) return callback(new Error('Invalid password'))
;

        const token = jwt.encode({
          username: username,
          expire: Date.now() + (1000 * 60 * 60) //1 hour
        }, tokenSecret);

        callback(null, token);
      });
    });
  };
```

```
  authService.checkToken = (token, callback) => {
    let userData;
    try {
      //jwt.decode will throw if the token is invalid
      userData = jwt.decode(token, tokenSecret);
      if(userData.expire <= Date.now()) {
        throw new Error('Token expired');
      }
    } catch(err) {
      return process.nextTick(callback.bind(null, err));
    }

    users.get(userData.username, (err, user) => {
      if (err) return callback(err);
      callback(null, {username: userData.username});
    });
  };

  return authService;
};
```

`authService` 模块将服务定位器作为输入的工厂。使用服务定位器的 `get()` 方法检索模块的两个依赖关系，即 `db` 对象和 `tokenSecret` （这是另一个配置参数）。

以类似的方式，我们可以转换 `lib/authController.js` 模块：

```
"use strict";

module.exports = (serviceLocator) => {
  const authService = serviceLocator.get('authService');
  const authController = {};

  authController.login = (req, res, next) => {
    authService.login(req.body.username, req.body.password,
      (err, result) => {
        if (err) {
          return res.status(401).send({
            ok: false,
            error: 'Invalid username/password'
          });
        }
        res.status(200).send({ok: true, token: result});
      }
    );
  };

  authController.checkToken = (req, res, next) => {
    authService.checkToken(req.query.token,
      (err, result) => {
        if (err) {
          return res.status(401).send({
            ok: false,
            error: 'Token is invalid or expired'
          });
        }
        res.status(200).send({ok: 'true', user: result});
      }
    );
  };

  return authController;
};
```

现在来看如何实例化和配置服务定位器。当然，这发生在 `app.js` 模块中：

```
"use strict";

const Express = require('express');
const bodyParser = require('body-parser');
const errorHandler = require('errorhandler');
const http = require('http');

const app = module.exports = new Express();
app.use(bodyParser.json());

const svcLoc = require('./lib/serviceLocator')();

svcLoc.register('dbName', 'example-db');
svcLoc.register('tokenSecret', 'SHHH!');
svcLoc.factory('db', require('./lib/db'));
svcLoc.factory('authService', require('./lib/authService'));
svcLoc.factory('authController', require('./lib/authController')
);

const authController = svcLoc.get('authController');

app.post('/login', authController.login);
app.get('/checkToken', authController.checkToken);

app.use(errorHandler());
http.createServer(app).listen(3000, () => {
  console.log('Express server started');
});
```

这就是新的服务定位器的连接方式：

1. 我们通过调用工厂实例化一个新的服务定位器。
2. 针对服务定位器注册配置参数和模块工厂。在这一点上，我们所有的依赖关系还没有实例化。我们只是注册他们的工厂。
3. 我们从服务定位器加载 `authController` ；这是在我们的应用程序的整个依赖关系图的实例化的入口点。当我们询问 `authController` 组件的实例时，服务定位器通过注入自己的一个实例来调用关联的工厂，然后 `authController` 工厂将尝试加载 `authService` 模块，然后实例化 `db` 模块。

服务定位器惰性加载模块。每个实例仅在需要时创建。还有另一个重要的含义：事实上，我们可以看到，每个依赖关系都是自动连接的，无需事先手动完成。好处是我们不必事先知道实例化和连接模块的正确顺序是什么 - 这一切都是自动和按需进行的。与简单的依赖注入模式相比，这更方便。

> 另一种常见模式是使用 `Express` 服务器实例作为简单的服务定位器。这可以通过使用 `expressApp.set(name，instance)` 来注册一个服务和 `expressApp.get(name)` 来获得。这种模式的一个很方便的地方就是作为

> 服务定位器的服务器实例已经被注入到每个中间件中，并且可以通过 `request.app` 属性来访问。可以在随处找到这个模式的例子。

服务定位器的优点和缺点

服务定位器和依赖注入具有很多共同点：都将依赖关系所有权转移到组件外部的实体。但是连接服务定位器的方式决定这个模式的灵活性。我们选择一个注入的服务定位器来实现我们的例子，而不是硬性的或全局的服务定位器，这几乎就是这种模式优势所在。实际上，结果将会是，我们不是使用 `require()` 将组件直接耦合到它的依赖项，而是将它耦合到服务定位器的一个特定实例。硬编码的服务定位器在配置与特定名称关联的组件时仍然具有更大的灵活性，但是在复用性方面仍然没有什么大的优势。

此外，与 `DI` 一样，使用服务定位器使得在运行时解决组件之间的关系变得更加困难。另外，这也使得我们更难准确知道特定组件的互相依赖。使用 `DI` ，可以用更清晰的方式表示：通过在工厂或构造函数参数中声明依赖关系。有了服务定位器，这个问题就不那么清楚了，需要在文档中进行代码检查或显式声明，以解释特定组件将要加载的依赖关系。

最后要知道，一个服务定位器经常被错误地认为是一个 `DI` 容器，因为它与依赖注入中心扮演相同的角色；然而，这两者之间有很大的差别。使用服务定位器，每个组件都明确地从服务定位器本身加载它的依赖关系。当使用DI容器时，组件与容器互不所知。

这两种方法之间的区别是显而易见的，原因有两个：

- 可重用性：依赖于服务定位器的组件不易重用，因为它要求系统中有一个服务定位器
- 可读性：正如我们已经说过的，服务定位器混淆了组件的依赖性要求

就可重用性而言，我们可以说服务定位器模式位于硬编码依赖关系和 `DI` 之间。在方便和简单方面，它肯定比手动 `DI` 更好，因为我们不必手动关心构建整个依赖关系图。

在这些假设下， `DI` 容器在组件的可重用性和便利性方面有更大的优势。我们将在下一节中更好地分析这种模式。

依赖注入容器

将服务定位器转换为依赖注入（ `DI` ）容器的步骤并不复杂，但正如我们已经提到的，它在解耦方面优势很大。事实上，每个模块都不需要依赖服务定位器，只需在依赖关系上表达需求， `DI` 容器就可以无缝地完成其他任务。正如我们将看到的，这个机制的优势在于，即使没有容器，每个模块都可以被重用。

向依赖注入容器声明一组依赖关系

依赖注入容器本质上是一个服务定位器，增加了一个功能：它在实例化之前标识模块的依赖性需求。为了做到这一点，一个模块必须以某种方式声明它的依赖关系，正如我们将看到的，我们有多种选择声明依赖关系。

第一种，也许是最流行的技术，是基于工厂或构造函数中使用的参数名称注入一组依赖关系。以 `authService` 模块为例：

```
module.exports = (db, tokenSecret) => {
  //...
}
```

正如我们所定义的，前面的模块将由我们的依赖注入容器使用名称为 `db` 和 `tokenSecret` 的依赖关系来实例化，这是一个非常简单直观的机制。 但是，为了能够读取函数参数的名称，有必要使用一些小技巧。在 `JavaScript` 中，我们有可能序列化一个函数，在运行时获取它的源代码; 这与在函数引用上调用 `toString()` 一样简单。用正则表达式，获取参数列表当然不是黑魔法。

> AngularJS是一个由 `Google` 开发的客户端 `JavaScript` 框架，它完全建立在DI容器之上，这种使用函数参数名称注入一组依赖关系的技术被广泛使用。

这种方法最大的问题是，源代码过长，这是一种在客户端 `JavaScript` 中广泛使用的做法，其中包括应用特定的代码转换以减小源代码的大小。有一种变量名称变更的技术，该技术基本上重命名任何局部变量以减少其长度，通常是单个字符。坏消息是函数参数是局部变量，通常会受到这个过程的影响，导致我们描述的声明依赖关系崩溃的机制。尽管在服务器端代码中缩小并不是非常必要，但重要的是要考虑到 `Node.js` 模块经常与浏览器共享，这是我们分析中需要考虑的一个重要因素。

幸运的是，依赖注入容器可能使用其他技术来知道要注入哪些依赖关系。这些技术如下：

- 我们可以使用附加到工厂函数的特殊属性，例如，显式列出要注入的所有依赖项的数组：

```
module.exports = (a, b) => {};
module.exports._inject = ['db', 'another/dependency'];
```

- 我们可以指定一个模块作为依赖项名称的数组，然后是工厂函数：

```
module.exports = ['db', 'another/depencency',(a, b) => {}];
```

- 我们可以使用附加到函数的每个参数的注释注释（但是，对于缩小源代码的体积，这也不能很好地发挥作用）：

```
module.exports = function(a /*db*/, b /*another/depencency*/) {}
;
```

所有这些技术都各有优势，因此对于我们的例子，我们将使用最简单和流行的方法，即使用函数的参数来获得依赖项名称。

使用DI容器重构鉴权服务器

为了演示 `DI` 容器如何比服务定位器的耦合性更低，我们现在要再次重构我们的认证服务器，为此我们将使用我们使用纯 `DI` 模式的版本作为起点。实际上，我们要做的只是保留 `app.js` 模块的所有组件，除了 `app.js` 模块，它将是负责初始化容器的模块。

但首先，我们需要实施我们的 `DI` 容器。 让我们通过在 `lib/` 目录下创建一个名为 `diContainer.js` 的新模块来实现这一点。这是它的最初部分：

```
"use strict";

const fnArgs = require('parse-fn-args');

module.exports = () => {
  const dependencies = {};
  const factories = {};
  const diContainer = {};

  diContainer.factory = (name, factory) => {
    factories[name] = factory;
  };

  diContainer.register = (name, dep) => {
    dependencies[name] = dep;
  };

  diContainer.get = (name) => {
    if (!dependencies[name]) {
      const factory = factories[name];
      dependencies[name] = factory &&
          diContainer.inject(factory);
      if (!dependencies[name]) {
        throw new Error('Cannot find module: ' + name);
      }
    }
    return dependencies[name];
  };

  diContainer.inject = (factory) => {
    const args = fnArgs(factory)
      .map(function(dependency) {
        return diContainer.get(dependency);
      });
    return factory.apply(null, args);
  };

  return diContainer;
};
```

`diContainer` 模块的第一部分在功能上与我们的服务定位器完全相同 以前见过。唯一显着的区别是：

- 我们需要一个名为args-list的新的 `npm` 模块，我们将使用它来提取函数参数的名称
- 这一次，我们不是直接调用模块工厂，而是依赖另一个名为 `inject()` 的 `diContainer` 模块的方法，它将解析模块的依赖关系并使用它来调用工厂。

`inject()` 是使DI容器与服务定位器不同的原因。其逻辑非常简单：

1. 我们使用 `parse-fn-args` 库从我们接收的工厂函数中提取参数列表作为输入。
2. 然后，我们将每个参数名称映射到使用 `get()` 方法检索到的相应的依赖项实例。
3. 最后，我们所要做的只是通过提供我们刚刚生成的依赖列表来调用工厂。 我们的 `diContainer` 就是这样，正如我们所看到的，它与服务定位器没有多大的区别，但是通过注入依赖来实例化模块的简单步骤与注入整个服务定位器相比有着巨大的差异。

为了完成认证服务器的重构，我们还需要调整 `app.js` 模块：

```
"use strict";

const Express = require('express');
const bodyParser = require('body-parser');
const errorHandler = require('errorhandler');
const http = require('http');

const app = module.exports = new Express();
app.use(bodyParser.json());

const diContainer = require('./lib/diContainer')();

diContainer.register('dbName', 'example-db');
diContainer.register('tokenSecret', 'SHHH!');
diContainer.factory('db', require('./lib/db'));
diContainer.factory('authService', require('./lib/authService'))
;
diContainer.factory('authController', require('./lib/authControl
ler'));

const authController = diContainer.get('authController');

app.post('/login', authController.login);
app.get('/checkToken', authController.checkToken);

app.use(errorHandler());
http.createServer(app).listen(3000, () => {
  console.log('Express server started');
});
```

正如我们所看到的，应用程序模块的代码与我们在上一节中用于初始化服务定位器的代码相同。我们还可以注意到，为了引导DI容器，并因此触发整个依赖图的加载，我们仍然需要通过调用 `diContainer.get('authController')` 将其用作服务定位器。之后，在 `DI` 容器中注册的每个模块将被自动实例化和连接。

DI容器的优点和缺点

假如我们的模块使用 `DI` 容器，他有着依赖注入模式大部分优点和缺点。特别是，耦合度更低和可测试性更强，但另一方面，它比单纯的依赖注入模式更复杂，因为我们的依赖关系在运行时解决。一个 `DI` 容器也与服务定位器模式共享许多属性，但是它有一个事实，即它不强制模块依赖除了它的实际依赖之外的任何额外的模块。这是一个巨大的优势，因为它允许每个模块甚至在没有DI容器的情况下使用，因为可以使用简单的手动注入。

这本质上就是我们在本节中演示的内容：我们使用了纯粹的 `DI` 模式的认证服务器的版本，然后在不修改任何组件（ `app` 模块除外）的情况下，我们能够自动地注入每个依赖。

> 在 `npm` 上，你可以找到很多DI容器 https://www.npmjs.org/search?q=dependency%20injection 。

书写插件

对于软件工程师而言，书写越少的代码越好，通过使用插件来对功能进行拓展。不幸的是，这并不是很容易，书写插件在时间，资源和复杂性方面都有成本。尽管如此，我们还是希望通过书写插件来对系统进行扩展，即使是仅仅针对于系统的某些部分。但就是在这一部分上，我们将要探索怎么书写插件，并关注两个问题：

- 将应用程序服务暴露给插件
- 将插件集成到应用程序中

把插件作为包

通常在 `Node.js` 中，应用程序的插件作为包安装到项目的 `node_modules` 目录中。这样做有两个好处。首先，我们可以利用 `npm` 的功能来分发插件并管理它的依赖关系。其次，一个包可以有自己的私有依赖关系图，这样可以减少依赖关系之间发生冲突和不兼容的可能性，而不是让插件使用父项目的依赖关系。

以下目录结构给出了一个包含两个作为包分发的插件的应用程序示例：

```
application
  '-- node_modules
      |-- pluginA
      '-- pluginB
```

在 `Node.js` 中，这是一个非常普遍的做法。 一些流行的例子是用它的中间件gulp，grunt，nodebb，express和docpad。

但是，使用包的好处不仅限于外部插件。事实上，一种流行的模式是通过将其组件包装到包中来构建整个应用程序，就好像它们是内部插件一样。因此，我们可以不用在应用程序的主包中组织模块，而是为每个大块功能创建一个单独的包，并将其安装到 `node_modules` 目录中。

一个包可以是私有的，不一定在公共 `npm` 可用。我们总是可以将私有组织信息设置到 `package.json` 中，以防止意外发布到 `npm` 。然后，我们可以将这些包提交到一个版本控制系统，比如 `git` ，或者利用一个私有的 `npm` 服务器与团队的其他人分享。

Advanced Asynchronous Recipes

几乎所有我们迄今为止看到的设计模式都可以被认为是通用的，并且适用于应用程序的许多不同的领域。但是，有一套更具体的模式，专注于解决明确的问题。我们可以调用这些模式。就像现实生活中的烹饪一样，我们有一套明确的步骤来实现预期的结果。当然，这并不意味着我们不能用一些创意来定制设计模式，以配合我们的客人的口味，对于书写 `Node.js` 程序来说是必要的。在本章中，我们将提供一些常见的解决方案来解决我们在日常 `Node.js` 开发中遇到的一些具体问题。这些模式包括以下内容：

- 异步引入模块并初始化
- 在高并发的应用程序中使用批处理和缓存异步操作的性能优化
- 运行与 `Node.js` 处理并发请求的能力相悖的阻塞事件循环的同步 `CPU` 绑定操作

异步引入模块并初始化

在 `Chapter2-Node.js Essential Patterns` 中，当我们讨论 `Node.js` 模块系统的基本属性时，我们提到了 `require()` 是同步的，并且 `module.exports` 也不能异步设置。

这是在核心模块和许多 `npm` 包中存在同步 `API` 的主要原因之一，是否同步加载会被作为一个 `option` 参数被提供，主要用于初始化任务，而不是替代异步 `API` 。

不幸的是，这并不总是可能的。同步 `API` 可能并不总是可用的，特别是对于在初始化阶段使用网络的组件，例如执行三次握手协议或在网络中检索配置参数。 许多数据库驱动程序和消息队列等中间件系统的客户端都是如此。

广泛适用的解决方案

我们举一个例子：一个名为 `db` 的模块，它将会连接到远程数据库。 只有在连接和与服务器的握手完成之后， `db` 模块才能够接受请求。在这种情况下，我们通常有两种选择：

- 在开始使用之前确保模块已经初始化，否则则等待其初始化。每当我们想要在异步模块上调用一个操作时，都必须完成这个过程：

```
const db = require('aDb'); //The async module
module.exports = function findAll(type, callback) {
  if (db.connected) { //is it initialized?
    runFind();
  } else {
    db.once('connected', runFind);
  }

  function runFind() {
    db.findAll(type, callback);
  };
};
```

- 使用依赖注入（ `Dependency Injection` ）而不是直接引入异步模块。通过这样做，我们可以延迟一些模块的初始化，直到它们的异步依赖被完全初始化。 这种技术将管理模块初始化的复杂性转移到另一个组件，通常是它的父模块。 在下面的例子中，这个组件是 `app.js` ：

```
// 模块app.js
const db = require('aDb'); // aDb是一个异步模块
const findAllFactory = require('./findAll');
db.on('connected', function() {
  const findAll = findAllFactory(db);
  // 之后再执行异步操作
});


// 模块findAll.js
module.exports = db => {
  //db 在这里被初始化
  return function findAll(type, callback) {
    db.findAll(type, callback);
  }
}
```

我们可以看出，如果所涉及的异步依赖的数量过多，第一种方案便不太适用了。

另外，使用 `DI` 有时也是不理想的，正如我们在 `Chapter7-Wiring Modules` 中看到的那样。在大型项目中，它可能很快变得过于复杂，尤其对于手动完成并使用异步初始化模块的情况下。如果我们使用一个设计用于支持异步初始化模块的 `DI` 容器，这些问题将会得到缓解。

但是，我们将会看到，还有第三种方案可以让我们轻松地将模块从其依赖关系的初始化状态中分离出来。

预初始化队列

将模块与依赖项的初始化状态分离的简单模式涉及到使用队列和命令模式。这个想法是保存一个模块在尚未初始化的时候接收到的所有操作，然后在所有初始化步骤完成后立即执行这些操作。

实现一个异步初始化的模块

为了演示这个简单而有效的技术，我们来构建一个应用程序。首先创建一个名为 `asyncModule.js` 的异步初始化模块：

```
const asyncModule = module.exports;

asyncModule.initialized = false;
asyncModule.initialize = callback => {
  setTimeout(() => {
    asyncModule.initialized = true;
    callback();
  }, 10000);
};

asyncModule.tellMeSomething = callback => {
  process.nextTick(() => {
    if(!asyncModule.initialized) {
      return callback(
        new Error('I don\'t have anything to say right now')
      );
    }
    callback(null, 'Current time is: ' + new Date());
  });
};
```

在上面的代码中， `asyncModule` 展现了一个异步初始化模块的设计模式。 它有一个 `initialize()` 方法，在 `10` 秒的延迟后，将初始化的 `flag` 变量设置为 `true` ，并通知它的回调调用（ `10` 秒对于真实应用程序来说是很长的一段时间了，但是对于具有互斥条件的应用来说可能会显得力不从心）。

另一个方法 `tellMeSomething()` 返回当前的时间，但是如果模块还没有初始化，它抛出产生一个异常。 下一步是根据我们刚刚创建的服务创建另一个模块。 我们设计一个简单的 `HTTP` 请求处理程序，在一个名为 `routes.js` 的文件中实现：

```
const asyncModule = require('./asyncModule');

module.exports.say = (req, res) => {
  asyncModule.tellMeSomething((err, something) => {
    if(err) {
      res.writeHead(500);
      return res.end('Error:' + err.message);
    }
    res.writeHead(200);
    res.end('I say: ' + something);
  });
};
```

在 `handler` 中调用 `asyncModule` 的 `tellMeSomething()` 方法，然后将其结果写入 `HTTP` 响应中。 正如我们所看到的那样，我们没有对 `asyncModule` 的初始化状态进行任何检查，这可能会导致问题。

现在，创建 `app.js` 模块，使用核心 `http` 模块创建一个非常基本的 `HTTP` 服务器：

```
const http = require('http');
const routes = require('./routes');
const asyncModule = require('./asyncModule');

asyncModule.initialize(() => {
  console.log('Async module initialized');
});

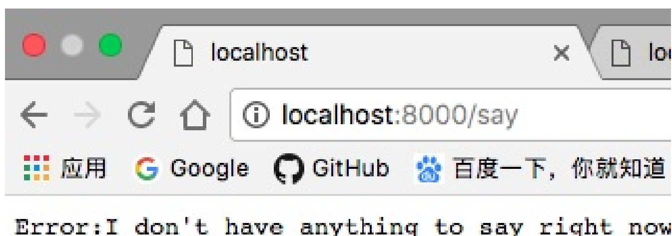
http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/say') {
    return routes.say(req, res);
  }
  res.writeHead(404);
  res.end('Not found');
}).listen(8000, () => console.log('Started'));
```

上述模块是我们应用程序的入口点，它所做的只是触发 `asyncModule` 的初始化并创建一个 `HTTP` 服务器，它使用我们以前创建的 `handler` （ `routes.say()` ）来对网络请求作出相应。

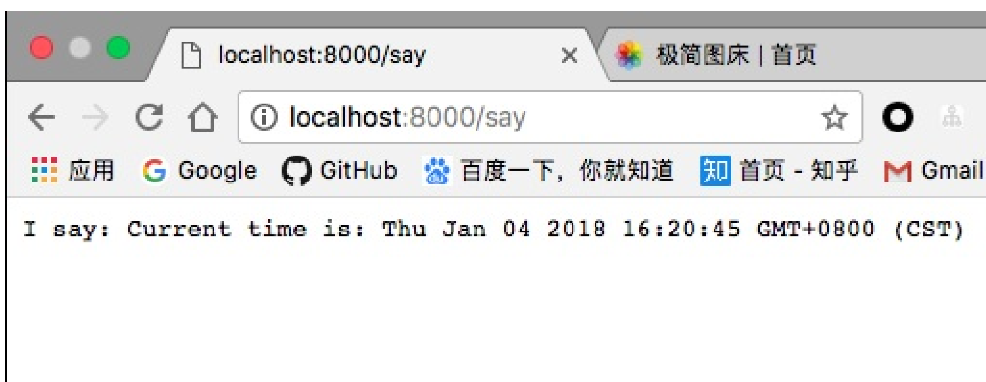
我们现在可以像往常一样通过执行 `app.js` 模块来尝试启动我们的服务器。

在服务器启动后，我们可以尝试使用浏览器访问 `URL` ： `http://localhost:8000/` 并查看从asyncModule返回的内容。 和预期的一样，如果我们在服务器启动后立即发送请求，结果将是一个错误，如下所示：

```
Error:I don't have anything to say right now
```



显然，在异步模块加载好了之后：



这意味着 `asyncModule` 尚未初始化，但我们仍尝试使用它，则会抛出一个错误。

根据异步初始化模块的实现细节，幸运的情况是我们可能会收到一个错误，乃至丢失重要的信息，崩溃整个应用程序。 总的来说，我们刚刚描述的情况总是必须要避免的。

大多数时候，可能并不会出现上述问题，毕竟初始化一般来说很快，以至于在实践中，它永远不会发生。 然而，对于设计用于自动调节的高负载应用和云服务器，情况就完全不同了。

用预初始化队列包装模块

为了维护服务器的健壮性，我们现在要通过使用我们在本节开头描述的模式来进行异步模块加载。我们将在 `asyncModule` 尚未初始化的这段时间内对所有调用的操作推入一个预初始化队列，然后在异步模块加载好后处理它们时立即刷新队列。这就是状态模式的一个很好的应用！我们将需要两个状态，一个在模块尚未初始化的时候将所有操作排队，另一个在初始化完成时将每个方法简单地委托给原始的 `asyncModule` 模块。

通常，我们没有机会修改异步模块的代码；所以，为了添加我们的排队层，我们需要围绕原始的 `asyncModule` 模块创建一个代理。

接下来创建一个名为 `asyncModuleWrapper.js` 的新文件，让我们依照每个步骤逐个构建它。我们需要做的第一件事是创建一个代理，并将原始异步模块的操作委托给这个代理：

```
const asyncModule = require('./asyncModule');
const asyncModuleWrapper = module.exports;
asyncModuleWrapper.initialized = false;
asyncModuleWrapper.initialize = () => {
  activeState.initialize.apply(activeState, arguments);
};
asyncModuleWrapper.tellMeSomething = () => {
  activeState.tellMeSomething.apply(activeState, arguments);
};
```

在前面的代码中， `asyncModuleWrapper` 将其每个方法简单地委托给 `activeState` 。让我们来看看这两个状态是什么样子

从 `notInitializedState` 开始， `notInitializedState` 是指还没初始化的状态：

```
// 当模块没有被初始化时的状态
let pending = [];
let notInitializedState = {

  initialize: function(callback) {
    asyncModule.initialize(function() {
      asyncModuleWrapper.initalized = true;
      activeState = initializedState;

      pending.forEach(function(req) {
        asyncModule[req.method].apply(null, req.args);
      });
      pending = [];

      callback();
    });
  },

  tellMeSomething: function(callback) {
    return pending.push({
      method: 'tellMeSomething',
      args: arguments
    });
  }

};
```

当 `initialize()` 方法被调用时，我们触发初始化 `asyncModule` 模块，提供一个回调函数作为参数。 这使我们的 `asyncModuleWrapper` 知道什么时候原始模块被初始化，在初始化后执行预初始化队列的操作，之后清空预初始化队列，再调用作为参数的回调函数，以下为具体步骤：

1. 把 `initializedState` 赋值给 `activeState` ，表示预初始化已经完成了。

2. 执行先前存储在待处理队列中的所有命令。
3. 调用原始回调。

由于此时的模块尚未初始化，此状态的 `tellMeSomething()` 方法仅创建一个新的 `Command` 对象，并将其添加到预初始化队列中。

此时，当原始的 `asyncModule` 模块尚未初始化时，代理应该已经清楚，我们的代理将简单地把所有接收到的请求防到预初始化队列中。 然后，当我们被通知初始化完成时，我们执行所有预初始化队列的操作，然后将内部状态切换到 `initializedState` 。来看这个代理模块最后的定义：

```
let initializedState = asyncModule;
```

不出意外， `initializedState` 对象只是对原始的 `asyncModule` 的引用！事实上，初始化完成后，我们可以安全地将任何请求直接发送到原始模块。

最后，设定异步模块还没加载好的的状态，即 `notInitializedState`

```
let activeState = notInitializedState;
```

我们现在可以尝试再次启动我们的测试服务器，但首先，我们不要忘记用我们新的 `asyncModuleWrapper` 对象替换原始的 `asyncModule` 模块的引用; 这必须在 `app.js` 和 `routes.js` 模块中完成。

这样做之后，如果我们试图再次向服务器发送一个请求，我们会看到在 `asyncModule` 模块尚未初始化的时候，请求不会失败; 相反，他们会挂起，直到初始化完成，然后才会被实际执行。我们当然可以肯定，比起之前，容错率变得更高了。

可以看到，在刚刚初始化异步模块的时候，服务器会等待请求的响应：

```
curl (curl)
➜ workspace curl localhost:8000/say
```

```
node (node)
^C
➜ test_code git:(master) ✗ node app.js
Started
```

在异步模块加载完成后，服务器才会返回响应的信息：

```
~/workspace (zsh)
➜ workspace curl localhost:8000/say
I say: Current time is: Thu Jan 04 2018 17:52:27 GMT+0800 (CST)%
➜ workspace
➜ workspace
```

```
..r09/test_code (zsh)
^C
➜ test_code git:(master) ✗ node app.js
Started
Async module initialized
^C
➜ test_code git:(master) ✗
```

> 模式：如果模块是需要异步初始化的，则对每个操作进行排队，直到模块完全初始化释放队列。

现在，我们的服务器可以在启动后立即开始接受请求，并保证这些请求都不会由于其模块的初始化状态而失败。我们能够在不使用 `DI` 的情况下获得这个结果，也不需要冗长且容易出错的检查来验证异步模块的状态。

其它场景的应用

我们刚刚介绍的模式被许多数据库驱动程序和 `ORM` 库所使用。 最值得注意的是Mongoose，它是 `MongoDB` 的 `ORM` 。使用 `Mongoose` ，不必等待数据库连接打开，以便能够发送查询，因为每个操作都排队，稍后与数据库的连接完全建立时执行。 这显然提高了其 `API` 的可用性。

> 看一下Mongoose的源码，它的每个方法是如何通过代理添加预初始化队列。可以看看实现这中模式的代码片段：https://github.com/Automattic/mongoose/blob/21f16c62e2f3230fe616745a40f22b4385a11b11/lib/drivers/node-mongodb-native/collection.js#L103-138

```
for (var i in Collection.prototype) {
  (function(i){
    NativeCollection.prototype[i] = function () {
      if (this.buffer) {
        // mongoose中，在缓冲区不为空时，只是简单地把这个操作加入缓冲区内
        this.addQueue(i, arguments);
        return;
      }

      var collection = this.collection
        , args = arguments
        , self = this
        , debug = self.conn.base.options.debug;

      if (debug) {
        if ('function' === typeof debug) {
          debug.apply(debug
            , [self.name, i].concat(utils.args(args, 0, args.len
gth-1)));
        } else {
          console.error('\x1B[0;36mMongoose:\x1B[0m %s.%s(%s) %s
 %s %s'
            , self.name
            , i
            , print(args[0])
            , print(args[1])
            , print(args[2])
            , print(args[3]))
        }
      }

      return collection[i].apply(collection, args);
    };
  })(i);
}
```

异步批处理和缓存

在高负载的应用程序中，缓存起着至关重要的作用，几乎在网络中的任何地方，从网页，图像和样式表等静态资源到纯数据（如数据库查询的结果）都会使用缓存。在本节中，我们将学习如何将缓存应用于异步操作，以及如何充分利用缓存解决高请求吞吐量的问题。

实现没有缓存或批处理的服务器

在这之前，我们来实现一个小型的服务器，以便用它来衡量缓存和批处理等技术在解决高负载应用程序的优势。

让我们考虑一个管理电子商务公司销售的 `web` 服务器，特别是对于查询我们的服务器所有特定类型的商品交易的总和的情况。 为此，考虑到 `LevelUP` 的简单性和灵活性，我们将再次使用 `LevelUP` 。我们要使用的数据模型是存储在 `sales` 这一个 `sublevel` 中的简单事务列表，它是以下的形式：

```
transactionId {amount, item}
```

`key` 由 `transactionId` 表示， `value` 则是一个 `JSON` 对象，它包含 `amount` ，表示销售金额和 `item` ，表示项目类型。 要处理的数据是非常基本的，所以以让我们立即在名为的 `totalSales.js` 文件中实现 `API` ，将如下所示：

```
const level = require('level');
const sublevel = require('level-sublevel');

const db = sublevel(level('example-db', {valueEncoding: 'json'})
);
const salesDb = db.sublevel('sales');

module.exports = function totalSales(item, callback) {
  console.log('totalSales() invoked');
  let sum = 0;
  salesDb.createValueStream()  // [1]
    .on('data', data => {
      if(!item || data.item === item) {  // [2]
        sum += data.amount;
      }
    })
    .on('end', () => {
      callback(null, sum);  // [3]
    });
};
```

该模块的核心是 `totalSales` 函数，它也是唯一 `exports` 的 `API` ；它进行如下工作：

1. 我们从包含交易信息的 `salesDb` 的 `sublevel` 创建一个 `Stream` 。 `Stream` 将从数据库中提取所有条目。

2. 监听 `data` 事件，这个事件触发时，将从数据库 `Stream` 中提取出每一项，如果这一项的 `item` 参数正是我们需要的 `item` ，就去累加它的 `amount` 到总的 `sum` 里面。
3. 最后， `end` 事件触发时，我们最终调用 `callback()` 方法。

上述查询方式可能在性能方面并不好。理想情况下，在实际的应用程序中，我们可以使用索引，甚至使用增量映射来缩短实时计算的时间；但是，由于我们需要体现缓存的优势，对于上述例子来说，慢速的查询实际上更好，因为它会突出显示我们要分析的模式的优点。

为了完成总销售应用程序，我们只需要从 `HTTP` 服务器公开 `totalSales` 的 `API` ；所以，下一步是构建一个（ `app.js` 文件）：

```
const http = require('http');
const url = require('url');
const totalSales = require('./totalSales');

http.createServer((req, res) => {
  const query = url.parse(req.url, true).query;
  totalSales(query.item, (err, sum) => {
    res.writeHead(200);
    res.end(`Total sales for item ${query.item} is ${sum}`);
  });
}).listen(8000, () => console.log('Started'));
```

我们创建的服务器是非常简单的；我们只需要它暴露 `totalSales API` 。 在我们第一次启动服务器之前，我们需要用一些示例数据填充数据库；我们可以使用专用于本节的代码示例中的 `populate_db.js` 脚本来执行此操作。该脚本将在数据库中创建 `100K` 个随机销售交易。 好的！ 现在，一切都准备好了。 像往常一样，启动服务器，我们执行以下命令：

```
node app
```

请求这个 `HTTP` 接口，访问至以下 `URL` ：

```
http://localhost:8000/?item=book
```

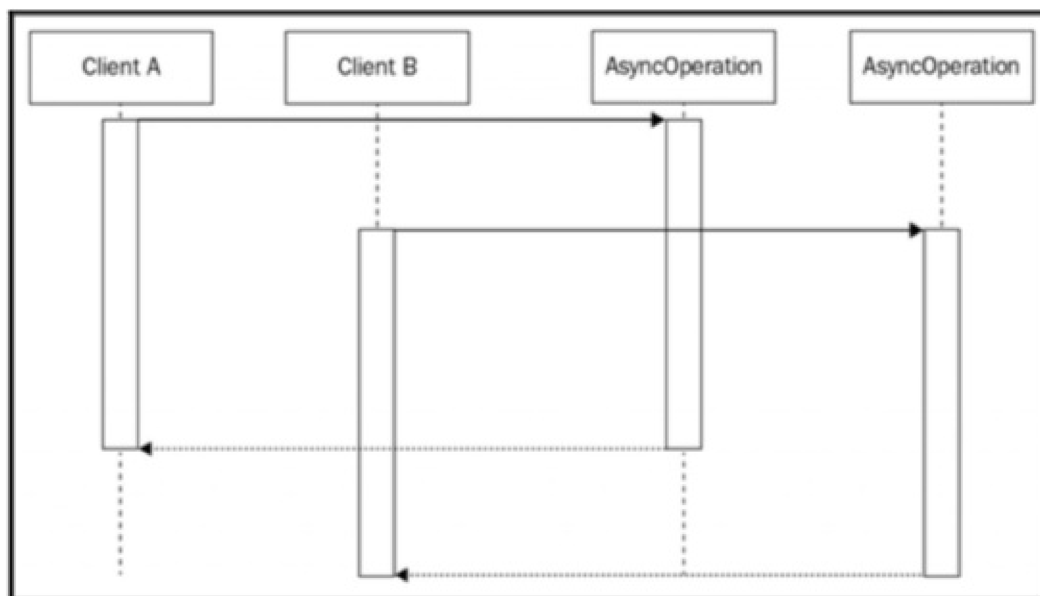
但是，为了更好地了解服务器的性能，我们需要连续发送多个请求;所以，我们创建一个名为 `loadTest.js` 的脚本，它以 `200 ms` 的间隔发送请求。它已经被配置为连接到服务器的 `URL` ，因此，要运行它，执行以下命令：

```
node loadTest
```

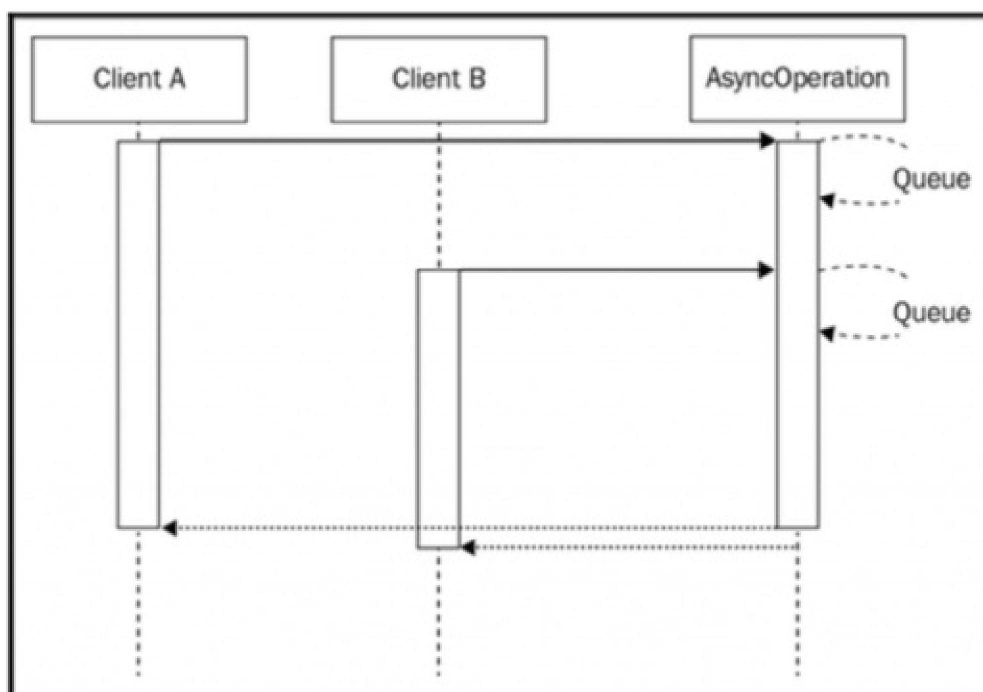
我们会看到这20个请求需要一段时间才能完成。注意测试的总执行时间，因为我们现在开始我们的服务，并测量我们可以节省多少时间。

批量异步请求

在处理异步操作时，最基本的缓存级别可以通过将一组调用集中到同一个 `API` 来实现。这非常简单：如果我们在调用异步函数的同时在队列中还有另一个尚未处理的回调，我们可以将回调附加到已经运行的操作上，而不是创建一个全新的请求。看下图的情况：



前面的图像显示了两个客户端（它们可以是两台不同的机器，或两个不同的 `Web` 请求），使用完全相同的输入调用相同的异步操作。 当然，描述这种情况的自然方式是由两个客户开始两个单独的操作，这两个操作将在两个不同的时刻完成，如前图所示。现在考虑下一个场景，如下图所示：



上图向我们展示了如何对 `API` 的两个请求进行批处理，或者换句话说，对两个请求执行到相同的操作。通过这样做，当操作完成时，两个客户端将同时被通知。这代表了一种简单而又非常强大的方式来降低应用程序的负载，而不必处理更复杂的缓存机制，这通常需要适当的内存管理和缓存失效策略。

在电子商务销售的Web服务器中使用批处理

现在让我们在 `totalSales API` 上添加一个批处理层。我们要使用的模式非常简单：如果在 `API` 被调用时已经有另一个相同的请求挂起，我们将把这个回调添加到一个队列中。当异步操作完成时，其队列中的所有回调立即被调用。

现在，让我们来改变之前的代码：创建一个名为 `totalSalesBatch.js` 的新模块。在这里，我们将在原始的 `totalSales API` 之上实现一个批处理层：

```
const totalSales = require('./totalSales');

const queues = {};
module.exports = function totalSalesBatch(item, callback) {
  if(queues[item]) {  // [1]
    console.log('Batching operation');
    return queues[item].push(callback);
  }

  queues[item] = [callback];  // [2]
  totalSales(item, (err, res) => {
    const queue = queues[item];  // [3]
    queues[item] = null;
    queue.forEach(cb => cb(err, res));
  });
};
```

`totalSalesBatch()` 函数是原始的 `totalSales() API` 的代理，它的工作原理如下：

1. 如果请求的 `item` 已经存在队列中，则意味着该特定 `item` 的请求已经在服务器任务队列中。在这种情况下，我们所要做的只是将回调 `push` 到现有队列，并立即从调用中返回。不进行后续操作。
2. 如果请求的 `item` 没有在队列中，这意味着我们必须创建一个新的请求。为此，我们为该特定 `item` 的请求创建一个新队列，并使用当前回调函数对其进行初始化。 接下来，我们调用原始的 `totalSales() API` 。
3. 当原始的 `totalSales()` 请求完成时，则执行我们的回调函数，我们遍历队列中为该特定请求的 `item` 添加的所有回调，并分别调用这些回调函数。

`totalSalesBatch()` 函数的行为与原始的 `totalSales() API` 的行为相同，不同之处在于，现在对于相同内容的请求 `API` 进行批处理，从而节省时间和资源。

想知道相比于 `totalSales() API` 原始的非批处理版本，在性能方面的优势是什么？然后，让我们将 `HTTP` 服务器使用的 `totalSales` 模块替换为我们刚刚创建的模块，修改 `app.js` 文件如下：

```
//const totalSales = require('./totalSales');
const totalSales = require('./totalSalesBatch');
http.createServer(function(req, res) {
// ...
});
```

如果我们现在尝试再次启动服务器并进行负载测试，我们首先看到的是请求被批量返回。

除此之外，我们观察到请求的总时间大大减少；它应该至少比对原始 `totalSales() API` 执行的原始测试快四倍！

这是一个惊人的结果，证明了只需应用一个简单的批处理层即可获得巨大的性能提升，比起缓存机制，也没有显得太复杂，因为，无需考虑缓存淘汰策略。

> 批处理模式在高负载应用程序和执行较为缓慢的 `API` 中发挥巨大作用，正是由于这种模式的运用，可以批量处理大量的请求。

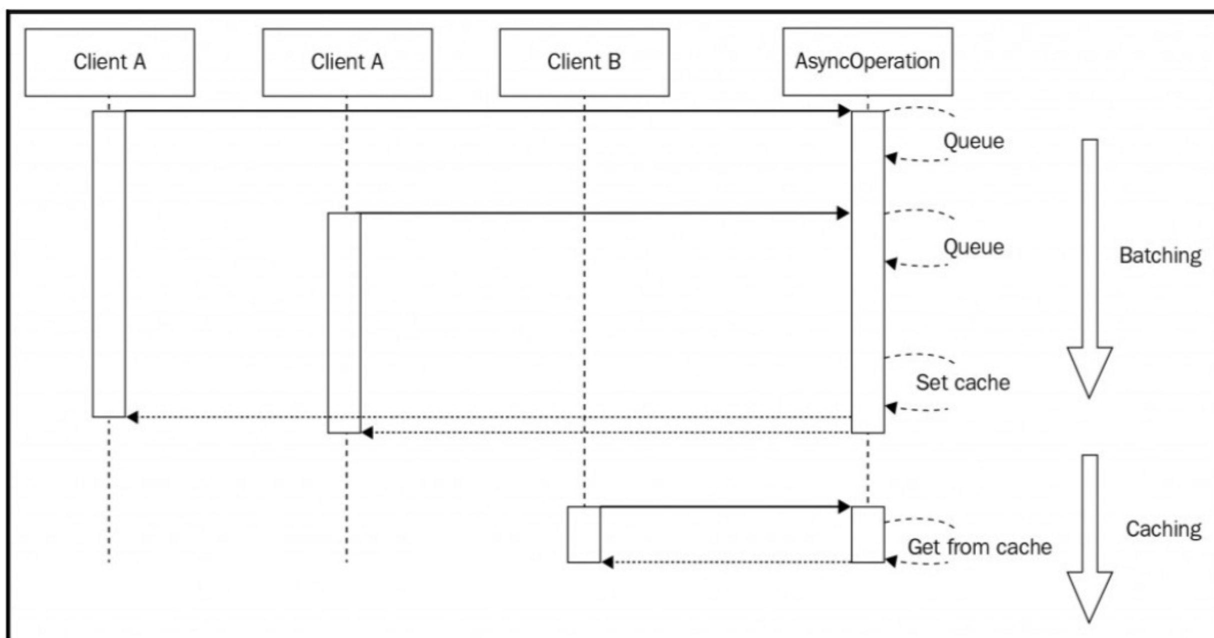
异步请求缓存策略

异步批处理模式的问题之一是对于 `API` 的答复越快，我们对于批处理来说，其意义就越小。有人可能会争辩说，如果一个 `API` 已经很快了，那么试图优化它就没有意义了。然而，它仍然是一个占用应用程序的资源负载的因素，总结起来，仍然可以有解决方案。另外，如果 `API` 调用的结果不会经常改变；因此，这时候批处理将并不会有较好的性能提升。在这种情况下，减少应用程序负载并提高响应速度的最佳方案肯定是更好的缓存模式。

缓存模式很简单：一旦请求完成，我们将其结果存储在缓存中，该缓存可以是变量，数据库中的条目，也可以是专门的缓存服务器。因此，下一次调用 `API` 时，可以立即从缓存中检索结果，而不是产生另一个请求。

对于一个有经验的开发人员来说，缓存不应该是多么新的技术，但是异步编程中这种模式的不同之处在于它应该与批处理结合在一起，以达到最佳效果。原因是因为多个请求可能并发运行，而没有设置缓存，并且当这些请求完成时，缓存将会被设置多次，这样做则会造成缓存资源的浪费。

基于这些假设，异步请求缓存模式的最终结构如下图所示：



上图给出了异步缓存算法的两个步骤：

1. 与批处理模式完全相同，与在未设置高速缓存时接收到的任何请求将一起批处理。这些请求完成时，缓存将会被设置一次。
2. 当缓存最终被设置时，任何后续的请求都将直接从缓存中提供。

另外我们需要考虑 `Zalgo` 的反作用（我们已经在 `Chapter 2-Node.js Essential Patterns` 中看到了它的实际应用）。在处理异步 `API` 时，我们必须确保始终以异步方式返回缓存的值，即使访问缓存只涉及同步操作。

在电子商务销售的Web服务器中使用异步缓存请求

实践异步缓存模式的优点，现在让我们将我们学到的东西应用到 `totalSales() API` 。

与异步批处理示例程序一样，我们创建一个代理，其作用是添加缓存层。

然后创建一个名为 `totalSalesCache.js` 的新模块，代码如下：

```
const totalSales = require('./totalSales');

const queues = {};
const cache = {};

module.exports = function totalSalesBatch(item, callback) {
  const cached = cache[item];
  if (cached) {
    console.log('Cache hit');
    return process.nextTick(callback.bind(null, null, cached));
  }

  if (queues[item]) {
    console.log('Batching operation');
    return queues[item].push(callback);
  }

  queues[item] = [callback];
  totalSales(item, (err, res) => {
    if (!err) {
      cache[item] = res;
      setTimeout(() => {
        delete cache[item];
      }, 30 * 1000); //30 seconds expiry
    }

    const queue = queues[item];
    queues[item] = null;
    queue.forEach(cb => cb(err, res));
  });
};
```

我们可以看到前面的代码与我们异步批处理的很多地方基本相同。 其实唯一的区别是以下几点：

- 我们需要做的第一件事就是检查缓存是否被设置，如果是这种情况，我们将立即使用 `callback()` 返回缓存的值，这里必须要使用 `process.nextTick()` ，因为缓存可能是异步设定的，需要等到下一次事件轮询时才能够保证缓存已经被设定。
- 继续异步批处理模式，但是这次，当原始 `API` 成功完成时，我们将结果保存到缓存中。此外，我们还设置了一个缓存淘汰机制，在 `30` 秒后使缓存失效。一个简单而有效的技术！

现在，我们准备尝试我们刚创建的 `totalSales` 模块。 先更改 `app.js` 模块，如下所示：

```
// const totalSales = require('./totalSales');
// const totalSales = require('./totalSalesBatch');
const totalSales = require('./totalSalesCache');
   http.createServer(function(req, res) {
      // ...
});
```

现在，重新启动服务器，并使用 `loadTest.js` 脚本进行配置，就像我们在前面的例子中所做的那样。使用默认的测试参数，与简单的异步批处理模式相比，很明显地有了更好的性能提升。 当然，这很大程度上取决于很多因素；例如收到的请求数量，以及一个请求和另一个请求之间的延迟等。当请求数量较高且跨越较长时间时，使用高速缓存批处理的优势将更为显著。

> **Memoization**被称做缓存函数调用的结果的算法。 在 `npm` 中，你可以找到许多包来实现异步的 `memoization` ，其中最著名的之一之一是memoizee。

有关实现缓存机制的说明

我们必须记住，在实际应用中，我们可能想要使用更先进的失效技术和存储机制。这可能是必要的，原因如下：

- 大量的缓存值可能会消耗大量内存。 在这种情况下，可以应用最近最少使用（ `LRU` ）算法来保持恒定的存储器利用率。
- 当应用程序分布在多个进程中时，对缓存使用简单变量可能会导致每个服务器实例返回不同的结果。如果这对于我们正在实现的特定应用程序来说是不希望的，那么解决方案就是使用共享存储来存储缓存。 常用的解决方案是Redis和Memcached。
- 与定时淘汰缓存相比，手动淘汰高速缓存可使得高速缓存使用寿命更长，同时提供更新的数据，但当然，管理起缓存来要复杂得多。

使用**Promise**进行批处理和缓存

在 `Chapter4-Asynchronous Control Flow Patterns with ES2015 and Beyond` 中，我们看到了 `Promise` 如何极大地简化我们的异步代码，但是在处理批处理和缓存时，它则可以提供更大的帮助。

利用 `Promise` 进行异步批处理和缓存策略，有如下两个优点：

- 多个 `then()` 监听器可以附加到相同的 `Promise` 实例。
- `then()` 监听器最多保证被调用一次，即使在 `Promise` 已经被 `resolve` 了之后， `then()` 也能正常工作。 此外， `then()` 总是会被保证其是异步调用的。

简而言之，第一个优点正是批处理请求所需要的，而第二个优点则在 `Promise` 已经是解析值的缓存时，也会提供同样的的异步返回缓存值的机制。

下面开始看代码，我们可以尝试使用 `Promises` 为 `totalSales()` 创建一个模块，在其中添加批处理和缓存功能。创建一个名为 `totalSalesPromises.js` 的新模块：

```
const pify = require('pify');  // [1]
const totalSales = pify(require('./totalSales'));

const cache = {};
module.exports = function totalSalesPromises(item) {
  if (cache[item]) {  // [2]
    return cache[item];
  }

  cache[item] = totalSales(item)  // [3]
    .then(res => {  // [4]
      setTimeout(() => {delete cache[item]}, 30 * 1000); //30 se
conds expiry
      return res;
    })
    .catch(err => {  // [5]
      delete cache[item];
      throw err;
    });
  return cache[item];  // [6]
};
```

`Promise` 确实很好，下面是上述函数的功能描述：

1. 首先，我们需要一个名为pify的模块，它允许我们对 `totalSales()` 模块进行 `promisification` 。这样做之后， `totalSales()` 将返回一个符合ES2015标准的 `Promise` 实例，而不是接受一个回调函数作为参数。
2. 当调用 `totalSalesPromises()` 时，我们检查给定的项目类型是否已经在缓存中有相应的 `Promise` 。如果我们已经有了这样的 `Promise` ，我们直接返回这个 `Promise` 实例。
3. 如果我们在缓存中没有针对给定项目类型的 `Promise` ，我们继续通过调用原始（ `promisified` ）的 `totalSales()` 来创建一个 `Promise` 实例。
4. 当 `Promise` 正常 `resolve` 了，我们设置了一个清除缓存的时间（假设为 `30秒` ），我们返回 `res` 将操作的结果返回给应用程序。
5. 如果 `Promise` 被异常 `reject` 了，我们立即重置缓存，并再次抛出错误，将其传播到 `Promise chain` 中，所以任何附加到相同 `Promise` 的其他应用程序也将收到这一异常。
6. 最后，我们返回我们刚才创建或者缓存的 `Promise` 实例。

非常简单直观，更重要的是，我们使用 `Promise` 也能够实现批处理和缓存。 如果我们现在要尝试使用 `totalSalesPromise()` 函数，稍微调整 `app.js` 模块，因为现在使用 `Promise` 而不是回调函数。 让我们通过创建一个名为 `appPromises.js` 的app模块来实现：

```
const http = require('http');
const url = require('url');
const totalSales = require('./totalSalesPromises');

http.createServer(function(req, res) {
  const query = url.parse(req.url, true).query;
  totalSales(query.item).then(function(sum) {
    res.writeHead(200);
    res.end(`Total sales for item ${query.item} is ${sum}`);
  });
}).listen(8000, function() {console.log('Started')});
```

它的实现与原始应用程序模块几乎完全相同，不同的是现在我们使用的是基于 `Promise` 的批处理/缓存封装版本; 因此，我们调用它的方式也略有不同。

运行以下命令开启这个新版本的服务器：

```
node appPromises
```

运行与CPU-bound的任务

虽然上面的 `totalSales()` 在系统资源上面消耗较大，但是其也不会影响服务器处理并发的能力。 我们在 `Chapter1-Welcome to the Node.js Platform` 中了解到有关事件循环的内容，应该为此行为提供解释：调用异步操作会导致堆栈退回到事件循环，从而使其免于处理其他请求。

但是，当我们运行一个长时间的同步任务时，会发生什么情况，从不会将控制权交还给事件循环？

这种任务也被称为 `CPU-bound` ，因为它的主要特点是 `CPU` 利用率较高，而不是 `I/O` 操作繁重。 让我们立即举一个例子上看看这些类型的任务在 `Node.js` 中的具体行为。

解决子集总和问题

现在让我们做一个 `CPU` 占用比较高的高计算量的实验。下面来看的是子集总和问题，我们计算一个数组中是否具有一个子数组，其总和为0。例如，如果我们有数组 `[1, 2, -4, 5, -3]` 作为输入，则满足问题的子数组是 `[1, 2, -3]` 和 `[2, -4, 5, -3]` 。

最简单的算法是把每一个数组元素做遍历然后依次计算，时间复杂度为 `O(2^n)` ，或者换句话说，它随着输入的数组长度成指数增长。这意味着一组 `20` 个整数则会有多达 `1, 048, 576` 中情况，显然不能够通过穷举来做到。当然，这个问题的解决方案可能并不算复杂。为了使事情变得更加困难，我们将考虑数组和问题的以下变化：给定一组整数，我们要计算所有可能的组合，其总和等于给定的任意整数。

```
const EventEmitter = require('events').EventEmitter;
class SubsetSum extends EventEmitter {
  constructor(sum, set) {
      super();
      this.sum = sum;
      this.set = set;
      this.totalSubsets = 0;
    } //...
}
```

`SubsetSum` 类是 `EventEmitter` 类的子类；这使得我们每次找到一个匹配收到的总和作为输入的新子集时都会发出一个事件。 我们将会看到，这会给我们很大的灵活性。

接下来，让我们看看我们如何能够生成所有可能的子集组合：

开始构建一个这样的算法。创建一个名为 `subsetSum.js` 的新模块。在其中声明一个 `SubsetSum` 类：

```
_combine(set, subset) {
  for(let i = 0; i < set.length; i++) {
    let newSubset = subset.concat(set[i]);
    this._combine(set.slice(i + 1), newSubset);
    this._processSubset(newSubset);
  }
}
```

不管算法其中到底是什么内容，但有两点要注意：

- `_combine()` 方法是完全同步的；它递归地生成每一个可能的子集，而不把 `CPU` 控制权交还给事件循环。如果我们考虑一下，这对于不需要任何 `I/O` 的算法来说是非常正常的。
- 每当生成一个新的组合时，我们都会将这个组合提供给 `_processSubset()` 方法以供进一步处理。

`_processSubset()` 方法负责验证给定子集的元素总和是否等于我们要查找的数字：

```
_processSubset(subset) {
  console.log('Subset', ++this.totalSubsets, subset);
  const res = subset.reduce((prev, item) => (prev + item), 0);
  if (res == this.sum) {
    this.emit('match', subset);
  }
}
```

简单地说，`_processSubset()` 方法将 `reduce` 操作应用于子集，以便计算其元素的总和。然后，当结果总和等于给定的 `sum` 参数时，会发出一个 `match` 事件。

最后，调用 `start()` 方法开始执行算法：

```
start() {
  this._combine(this.set, []);
  this.emit('end');
}
```

通过调用 `_combine()` 触发算法，最后触发一个 `end` 事件，表明所有的组合都被检查过，并且任何可能的匹配都已经被计算出来。 这是可能的，因为 `_combine()` 是同步的; 因此，只要前面的函数返回， `end` 事件就会触发，这意味着所有的组合都被计算出来了。

接下来，我们在网络上公开刚刚创建的算法。可以使用一个简单的 `HTTP` 服务器对响应的任务作出响应。 特别是，我们希望以 `/subsetSum?data=<Array>&sum=<Integer>` 这样的请求格式进行响应，传入给定的数组和 `sum` ，使用 `SubsetSum` 算法进行匹配。

在一个名为 `app.js` 的模块中实现这个简单的服务器：

```
const http = require('http');
const SubsetSum = require('./subsetSum');

http.createServer((req, res) => {
  const url = require('url').parse(req.url, true);
  if(url.pathname === '/subsetSum') {
    const data = JSON.parse(url.query.data);
    res.writeHead(200);
    const subsetSum = new SubsetSum(url.query.sum, data);
    subsetSum.on('match', match => {
      res.write('Match: ' + JSON.stringify(match) + '\n');
    });
    subsetSum.on('end', () => res.end());
    subsetSum.start();
  } else {
    res.writeHead(200);
    res.end('I\m alive!\n');
  }
}).listen(8000, () => console.log('Started'));
```

由于 `SubsetSum` 实例使用事件返回结果，所以我们可以在算法生成后立即对匹配的结果使用 `Stream` 进行处理。另一个需要注意的细节是，每次我们的服务器都会返回 `I'm alive!` ，这样我们每次发送一个不同于 `/subsetSum` 的请求的时候。可以用来检查我们服务器是否挂掉了，这在稍后将会看到。

开始运行：

```
node app
```

一旦服务器启动，我们准备发送我们的第一个请求；让我们尝试发送一组17个随机数，这将导致产生 `131,071` 个组合，那么服务器将会处理一段时间：

```
curl -G http://localhost:8000/subsetSum --data-urlencode "data=[
116,119,101,101,-116,109,101,-105,-102,117,-115,-97,119,-116,-10
4,-105,115]"--data-urlencode "sum=0"
```

这是如果我们在第一个请求仍在运行的时候在另一个终端中尝试输入以下命令，我们将发现一个巨大的问题：

```
curl -G http://localhost:8000
```

```
Subset 97255 [ 119, 101, 119, -116, 115 ]
Subset 97256 [ 119, 101, 119, -116 ]
Subset 97257 [ 119, 101, 119, -104, -105, 115 ]
Subset 97258 [ 119, 101, 119, -104, -105 ]
Subset 97259 [ 119, 101, 119, -104, 115 ]
Subset 97260 [ 119, 101, 119, -104 ]
Subset 97261 [ 119, 101, 119, -105, 115 ]
Subset 97262 [ 119, 101, 119, -105 ]
Subset 97263 [ 119, 101, 119, 115 ]
Subset 97264 [ 119, 101, 119 ]
Subset 97265 [ 119, 101, -116, -104, -105, 115 ]
Subset 97266 [ 119, 101, -116, -104, -105 ]
Subset 97267 [ 119, 101, -116, -104, 115 ]
Subset 97268 [ 119, 101, -116, -104 ]
Subset 97269 [ 119, 101, -116, -105, 115 ]
Subset 97270 [ 119, 101, -116, -105 ]
Subset 97271 [ 119, 101, -116, 115 ]
Subset 97272 [ 119, 101, -116 ]
Subset 97273 [ 119, 101, -104, -105, 115 ]
Subset 97274 [ 119, 101, -104, -105 ]
Subset 97275 [ 119, 101, -104, 115 ]
Subset 97276 [ 119, 101, -104 ]
Subset 97277 [ 119, 101, -105, 115 ]
```

```
➜ test_code git:(master) ✗ curl -G http://localhost:8000/subsetSum --data-urlencode "data=[116,
dquote>     119,101,101,-116,109,101,-105,-102,117,-115,-97,119,-116,-104,-105,115]" --
➜ test_code git:(master) ✗    data-urlencode "sum=0"
zsh: command not found: data-urlencode
➜ test_code git:(master) ✗    data-urlencode "sum=0"
zsh: command not found: data-urlencode
➜ test_code git:(master) ✗ curl -G http://localhost:8000/subsetSum --data-urlencode "data=[116,
    119,101,101,-116,109,101,-105,-102,117,-115,-97,119,-116,-104,-105,115]" --
    data-urlencode "sum=0"
```

```
➜ test_code git:(master) ✗    curl -G http://localhost:8000
```

我们会看到直到第一个请求结束之前，最后一个请求一直处于挂起的状态。服务器没有返回响应！这正如我们所想的那样。 `Node.js` 事件循环运行在一个单独的线程中，如果这个线程被一个长的同步计算阻塞，它将不能再执行一个循环来响应 `I'm alive!` ， 我们必须知道，这种代码显然不能够用于同时接收到多个请求的应用程序。

但是不要对 `Node.js` 中绝望，我们可以通过几种方式来解决这种情况。我们来分析一下最常见的两种方案：

使用setImmediate

通常， `CPU-bound` 算法是建立在一定规则之上的。它可以是一组递归调用，一个循环，或者基于这些的任何变化/组合。 所以，对于我们的问题，一个简单的解决方案就是在这些步骤完成后（或者在一定数量的步骤之后），将控制权交还给事件循环。这样，任何待处理的 `I / O` 仍然可以在事件循环在长时间运行的算法产

生 `CPU` 的时间间隔中处理。对于这个问题而言，解决这一问题的方式是把算法的下一步在任何可能导致挂起的 `I/O` 请求之后运行。这听起来像是 `setImmediate()` 方法的完美用例（我们已经在 `Chapter2-Node.js Essential Patterns` 中介绍过这一 `API` ）。

> 模式：使用 `setImmediate()` 交错执行长时间运行的同步任务。

使用setImmediate进行子集求和算法的步骤

现在我们来看看这个模式如何应用于子集求和算法。 我们所要做的只是稍微修改一下 `subsetSum.js` 模块。 为方便起见，我们将创建一个名为 `subsetSumDefer.js` 的新模块，将原始的 `subsetSum` 类的代码作为起点。 我们要做的第一个改变是添加一个名为 `_combineInterleaved()` 的新方法，它是我们正在实现的模式的核心：

```
_combineInterleaved(set, subset) {
  this.runningCombine++;
  setImmediate(() => {
    this._combine(set, subset);
    if(--this.runningCombine === 0) {
      this.emit('end');
    }
  });
}
```

正如我们所看到的，我们所要做的只是使用 `setImmediate()` 调用原始的同步的 `_combine()` 方法。然而，现在的问题是因为该算法不再是同步的，我们更难以知道何时已经完成了所有的组合的计算。

为了解决这个问题，我们必须使用非常类似于我们在 `Chapter3-Asynchronous Control Flow Patterns with Callbacks` 看到的异步并行执行的模式来追溯 `_combine()` 方法的所有正在运行的实例。当 `_combine()` 方法的所有实例都已经完成运行时，触发 `end` 事件，通知任何监听器，进程需要做的所有动作都已经完成。

对于最终子集求和算法的重构版本。首先，我们需要将 `_combine()` 方法中的递归步骤替换为异步：

```
_combine(set, subset) {
  for(let i = 0; i < set.length; i++) {
    let newSubset = subset.concat(set[i]);
    this._combineInterleaved(set.slice(i + 1), newSubset);
    this._processSubset(newSubset);
  }
}
```

通过上面的更改，我们确保算法的每个步骤都将使用 `setImmediate()` 在事件循环中排队，在事件循环队列中 `I / O` 请求之后执行，而不是同步运行造成阻塞。

另一个小调整是对于 `start()` 方法：

```
start() {
  this.runningCombine = 0;
  this._combineInterleaved(this.set, []);
}
```

在前面的代码中，我们将 `_combine()` 方法的运行实例的数量初始化为 `0`.我们还通过调用 `_combineInterleaved()` 来将调用替换为 `_combine()` ，并移除了 `end` 的触发，因为现在 `_combineInterleaved()` 是异步处理的。 通过这个最后的改变，我们的子集求和算法现在应该能够通过事件循环可以运行的时间间隔交替地运行其可能大量占用 `CPU` 的代码，并且不会再造成阻塞。

最后更新 `app.js` 模块，以便它可以使用新版本的 `SubsetSum` ：

```
const http = require('http');
// const SubsetSum = require('./subsetSum');
const SubsetSum = require('./subsetSumDefer');
http.createServer(function(req, res) {
  // ...
})
```

和之前一样的方式开始运行,结果如下：

此时，使用异步的方式运行，不再会阻塞 `CPU` 了。

interleaving模式

正如我们所看到的，在保持应用程序的响应性的同时运行一个 `CPU-bound` 的任务并不复杂，只需要使用 `setImmediate()` 把同步执行的代码变为异步执行即可。但是，这不是效率最好的模式；实际上，延迟执行一个任务会额外带来一个小的开销，在这样的算法中，积少成多，则会产生重大的影响。这通常是我们在运行 `CPU` 限制任务时所需要的最后一件事情，特别是如果我们必须将结果直接返回给用户，这应该在合理的时间内进行响应。 缓解这个问题的一个可能的解决方案是只有在一定数量的步骤之后使用 `setImmediate()` ，而不是在每一步中使用它。但是这仍然不能解决问题的根源。

记住，这并不是说一旦我们想要通过异步的模式来执行 `CPU-bound` 的任务，我们就应该不惜一切代价来避免这样的额外开销，事实上，从更广阔的角度来看，同步任务并不一定非常漫长和复杂，以至于造成麻烦。在繁忙的服务器中，即使是阻塞事件循环 `200` 毫秒的任务也会产生不希望的延迟。 在那些并发量并不高的服务器来说，即使产生一定短时的阻塞，也不会影响性能，使用交错执行 `setImmediate()` 可能是避免阻塞事件循环的最简单也是最有效的方法。

> `process.nextTick()` 不能用于交错长时间运行的任务。正如我们在 `Chapter1-Welcome to the Node.js Platform` 中看到的， `nextTick()` 会在任何未返回的 `I / O` 之前调度，并且在重复调用 `process.nextTick()` 最终会导致 `I / O` 饥饿。 你可以通过在前面的例子中用 `process.nextTick()` 替换 `setImmediate()` 来验证。

使用多个进程

使用 `interleaving模式` 并不是我们用来运行 `CPU-bound` 任务的唯一方法；防止事件循环阻塞的另一种模式是使用子进程。我们已经知道 `Node.js` 在运行 `I / O` 密集型应用程序（如Web服务器）的时候是最好的，因为 `Node.js` 可以使得我们可以通过异步来优化资源利用率。

所以，我们必须保持应用程序响应的最好方法是不要在主应用程序的上下文中运行昂贵的 `CPU-bound` 任务，而是使用单独的进程。这有三个主要的优点：

- 同步任务可以全速运行，而不需要交错执行的步骤
- 在 `Node.js` 中处理进程很简单，可能比修改一个使用 `setImmediate()` 的算法更容易，并且多进程允许我们轻松使用多个处理器，而无需扩展主应用程序本身。
- 如果我们真的需要超高的性能，可以使用低级语言，如性能良好的 `C` 。

`Node.js` 有一个充足的 `API` 库带来与外部进程交互。 我们可以在 `child_process` 模块中找到我们需要的所有东西。 而且，当外部进程只是另一个 `Node.js` 程序时，将它连接到主应用程序是非常容易的，我们甚至不觉得我们在本地应用程序外部运行任何东西。这得益于 `child_process.fork()` 函数，该函数创建一个新的子 `Node.js` 进程，并自动创建一个通信管道，使我们能够使用与 `EventEmitter` 非常相似的接口交换信息。来看如何用这个特性来重构我们的子集求和算法。

将子集求和任务委托给其他进程

重构 `SubsetSum` 任务的目标是创建一个单独的子进程，负责处理 `CPU-bound` 的任务，使服务器的事件循环专注于处理来自网络的请求：

1. 我们将创建一个名为 `processPool.js` 的新模块，它将允许我们创建一个正在运行的进程池。创建一个新的进程代价昂贵，需要时间，因此我们需要保持它们不断运行，尽量不要产生中断，时刻准备好处理请求，使我们可以节省时间和 `CPU` 。此外，进程池需要帮助我们限制同时运行的进程数量，以避免将使我们的应用程序受到拒绝服务（ `DoS` ）攻击。
2. 接下来，我们将创建一个名为 `subsetSumFork.js` 的模块，负责抽象子进程中运行的 `SubsetSum` 任务。 它的角色将与子进程进行通信，并将任务的结果展示为来自当前应用程序。
3. 最后，我们需要一个 `worker` （我们的子进程），一个新的 `Node.js` 程序，运行子集求和算法并将其结果转发给父进程。

> DoS攻击是企图使其计划用户无法使用机器或网络资源，例如临时或无限中断或暂停连接到Internet的主机的服务。

实现一个进程池

先从构建 `processPool.js` 模块开始：

```
const fork = require('child_process').fork;
class ProcessPool {
  constructor(file, poolMax) {
      this.file = file;
      this.poolMax = poolMax;
      this.pool = [];
      this.active = [];
      this.waiting = [];
    } //...
}
```

在模块的第一部分，引入我们将用来创建新进程的 `child_process.fork()` 函数。 然后，我们定义 `ProcessPool` 的构造函数，该构造函数接受表示要运行的 `Node.js` 程序的文件参数以及池中运行的最大实例数 `poolMax` 作为参数。然后我们定义三个实例变量：

- `pool` 表示的是准备运行的进程
- `active` 表示的是当前正在运行的进程列表
- `waiting` 包含所有这些请求的任务队列，保存由于缺少可用的资源而无法立即实现的任务

看 `ProcessPool` 类的 `acquire()` 方法，它负责取出一个准备好被使用的进程：

```
acquire(callback) {
  let worker;
  if(this.pool.length > 0) {  // [1]
    worker = this.pool.pop();
    this.active.push(worker);
    return process.nextTick(callback.bind(null, null, worker));
  }

  if(this.active.length >= this.poolMax) {  // [2]
    return this.waiting.push(callback);
  }

  worker = fork(this.file);  // [3]
  this.active.push(worker);
  process.nextTick(callback.bind(null, null, worker));
}
```

函数逻辑如下：

1. 如果在进程池中有一个准备好被使用的进程，我们只需将其移动到 `active` 数组中，然后通过异步的方式调用其回调函数。
2. 如果池中没有可用的进程，或者已经达到运行进程的最大数量，必须等待。通过把当前回调放入 `waiting` 数组。
3. 如果我们还没有达到运行进程的最大数量，我们将使用 `child_process.fork()` 创建一个新的进程，将其添加到 `active` 列表中，然后调用其回调。

`ProcessPool` 类的最后一个方法是 `release()` ，其目的是将一个进程放回进程池中：

```
release(worker) {
  if(this.waiting.length > 0) {  // [1]
    const waitingCallback = this.waiting.shift();
    waitingCallback(null, worker);
  }
  this.active = this.active.filter(w => worker !==  w);  // [2]
  this.pool.push(worker);
}
```

前面的代码也很简单，其解释如下：

- 如果在 `waiting` 任务队列里面有任务需要被执行，我们只需为这个任务分配一个进程 `worker` 执行。
- 否则，如果在 `waiting` 任务队列中都没有需要被执行的任务，我们则把 `active` 的进程列表中的进程放回进程池中。

正如我们所看到的，进程从来没有中断，只在为其不断地重新分配任务，使我们可以通过在每个请求不重新启动一个进程达到节省时间和空间的目的。然而，重要的是要注意，这可能并不总是最好的选择，这很大程度上取决于我们的应用程序的要求。为减少进程池长期占用内存，可能的调整如下：

- 在一个进程空闲一段时间后，终止进程，释放内存空间。
- 添加一个机制来终止或重启没有响应的或者崩溃了的进程。

父子进程通信

现在我们的 `ProcessPool` 类已经准备就绪，我们可以使用它来实现 `SubsetSumFork` 模块， `SubsetSumFork` 的作用是与子进程进行通信得到子集求和的结果。前面曾说到，用 `child_process.fork()` 启动一个进程也给了我们创建了一个简单的基于消息的管道，通过实现 `subsetSumFork.js` 模块来看看它是如何工作的：

```
const EventEmitter = require('events').EventEmitter;
const ProcessPool = require('./processPool');
const workers = new ProcessPool(__dirname + '/subsetSumWorker.js'
, 2);

class SubsetSumFork extends EventEmitter {
  constructor(sum, set) {
    super();
    this.sum = sum;
    this.set = set;
  }

  start() {
    workers.acquire((err, worker) => {  // [1]
      worker.send({sum: this.sum, set: this.set});

      const onMessage = msg => {
        if (msg.event === 'end') {  // [3]
          worker.removeListener('message', onMessage);
          workers.release(worker);
        }

        this.emit(msg.event, msg.data);  // [4]
      };

      worker.on('message', onMessage);  // [2]
    });
  }
}

module.exports = SubsetSumFork;
```

首先注意，我们在 `subsetSumWorker.js` 调用 `ProcessPool` 的构造函数创建 `ProcessPool` 实例。 我们还将进程池的最大容量设置为 `2` 。

另外，我们试图维持原来的 `SubsetSum` 类相同的公共API。实际上， `SubsetSumFork` 是 `EventEmitter` 的子类，它的构造函数接受 `sum` 和 `set` ，而 `start()` 方法则触发算法的执行，而这个 `SubsetSumFork` 实例运行在一个单独的进程上。调用 `start()` 方法时会发生的情况：

1. 我们试图从进程池中获得一个新的子进程。在创建进程成功之后，我们尝试向子进程发送一条消息，包含 `sum` 和 `set` 。 `send()` 方法是 `Node.js` 自动提供给 `child_process.fork()` 创建的所有进程，这实际上与父子进程之间的通信管道有关。
2. 然后我们开始监听子进程返回的任何消息，我们使用 `on()` 方法附加一个新的事件监听器（这也是所有以 `child_process.fork()` 创建的进程提供的通信通道的一部分）。
3. 在事件监听器中，我们首先检查是否收到一个 `end` 事件，这意味着 `SubsetSum` 所有任务已经完成，在这种情况下，我们删除 `onMessage` 监听器并释放 `worker` ，并将其放回进程池中，不再让其占用内存资源和 `CPU` 资源。
4. `worker` 以 `{event，data}` 格式生成消息，使得任何时候一旦子进程处理完毕任务，我们在外部都能接收到这一消息。

这就是 `SubsetSumFork` 模块现在我们来实现这个 `worker` 应用程序。

与父进程进行通信

现在我们来创建 `subsetSumWorker.js` 模块，我们的应用程序，这个模块的全部内容将在一个单独的进程中运行：

```
const SubsetSum = require('./subsetSum');

process.on('message', msg => {  // [1]
  const subsetSum = new SubsetSum(msg.sum, msg.set);

  subsetSum.on('match', data => {  // [2]
    process.send({event: 'match', data: data});
  });

  subsetSum.on('end', data => {
    process.send({event: 'end', data: data});
  });

  subsetSum.start();
});
```

由于我们的 `handler` 处于一个单独的进程中，我们不必担心这类 `CPU-bound` 任务阻塞事件循环，所有的 `HTTP` 请求将继续由主应用程序的事件循环处理，而不会中断。

当子进程开始启动时，父进程：

1. 子进程立即开始监听来自父进程的消息。这可以通过 `process.on()` 函数轻松实现。我们期望从父进程中唯一的消息是为新的 `SubsetSum` 任务提供输入的消息。只要收到这样的消息，我们创建一个 `SubsetSum` 类的新实例，并注册 `match` 和 `end` 事件监听器。最后，我们用 `subsetSum.start()` 开始计算。
2. 每次子集求和算法收到事件时，把结果它封装在格式为 `{event，data}` 的对象中，并将其发送给父进程。这些消息然后在 `subsetSumFork.js` 模块中处理，就像我们在前面的章节中看到的那样。

> 注意：当子进程不是 `Node.js` 进程时，则上述的通信管道就不可用了。在这种情况下，我们仍然可以通过在暴露于父进程的标准输入流和标准输出流之上实现我们自己的协议来建立父子进程通信的接口。

多进程模式

尝试新版本的子集求和算法，我们只需要替换 `HTTP` 服务器使用的模块（文件 `app.js` ）：

运行结果如下：

更有趣的是，我们也可以尝试同时启动两个 `subsetSum` 任务，我们可以充分看到多核 `CPU` 的作用。 相反，如果我们尝试同时运行三个 `subsetSum` 任务，结果应该是最后一个启动将挂起。这不是因为主进程的事件循环被阻塞，而是因为我们为 `subsetSum` 任务设置了两个进程的并发限制。

正如我们所看到的，多进程模式比interleaving模式更加强大和灵活；然而，由于单个机器提供的 `CPU` 和内存资源量仍然是一个硬性限制，所以它仍然不可扩展。在这种情况下，将负载分配到多台机器上，则是更优秀的解决办法。

> 值得一提的是，在运行`CPU-bound`任务时，多线程可以成为多进程的替代方案。目前，有几个`npm`包公开了一个用于处理用户级模块的线程的`API`；其中最流行的是webworker-threads。但是，即使线程更轻量级，完整的进程也可以提供更大的灵活性，并具备更高更可靠的容错处理。

总结

本章讲述以下三点：

- 异步初始化模块
- 批处理和缓存在`Node.js`异步中的运用
- 使用异步或者多进程来处理`CPU-bound`的任务

Scalability and Architectural Patterns

在早期， `Node.js` 主要用于非阻塞的 `Web` 服务器，它的原名实际上是 `web.js` 。其创建者 `Ryan Dahl` 很快意识到了该平台的潜力，并开始使用工具对其进行扩展，以便在 `JavaScript / non-blocking paradigm` 之上创建任何类型的服务器端应用程序。 `Node.js` 的特点对于分布式系统的实现是完美的，由分布式系统组成的节点通过网络协调运作。 `Node.js` 诞生了。与其他网络平台不同的是，非阻塞这个词在应用程序的生命周期很早就进入了 `Node.js` 开发者的词汇表中，主要是因为它具有单线程特性，不能利用机器的所有资源，但通常还有更多深刻的原因。正如我们将在本章中看到的，扩展应用程序不仅意味着增加其容量，使其能够更快地处理更多的请求；这也是实现高可用性和容错性的关键途径。令人惊讶的是，它也可以将应用程序的复杂性分解为更易于管理的部分。可伸缩性是一个具有多个面的概念，其中六个是精确的，就像一个立方体的面 - 即多维数据集的面。

在本章中，我们将学习以下主题：

- `scale cube` 是什么
- 如何通过运行同一应用程序的多个实例来进行扩展
- 如何在扩展应用程序时利用负载平衡器
- 什么是服务注册表，以及如何使用它
- 如何从单片应用程序设计微服务架构
- 如何通过使用一些简单的架构模式来集成大量的服务

介绍应用程序缩放

在我们深入探讨一些实际的模式和例子之前，我们应该说一下应用程序扩展的原因以及如何实现。

缩放 `Node.js` 应用程序

我们已经知道，典型的 `Node.js` 应用程序的大部分任务都是在单个线程的上下文中运行的。在 `Chapter1-Welcome to the Node.js Platform` ，我们了解到这不是一个限制，而是一个优点，因为它允许应用程序优化处理并发请求所需资源的使用情况，这要归功于非阻塞 `I / O` 范例。由非阻塞 `I / O` 充分利用的单线程对于每秒处理中等数量的请求（通常每秒几百次（这在很大程度上取决于应用程序））的应用程序奇妙地工作。假设我们使用的是商品硬件，那么无论服务器的功能如何强大，单个线程所能支持的容量都是有限的，因此，如果我们想要将 `Node.js` 用于高负载应用程序，唯一的方法就是将其扩展多个进程和机器。 但是，工作负载不是缩放 `Node.js` 应用程序的唯一原因;事实上，使用相同的技术，我们可以获得其他所需的属性，如可用性和容错性。可伸缩性也是适用于应用程序的大小和复杂性的概念;实际上，可拓展性是设计软件的另一个重要因素。 `JavaScript` 是一个谨慎使用的工具，缺乏类型检查和许多陷阱可能是应用程序

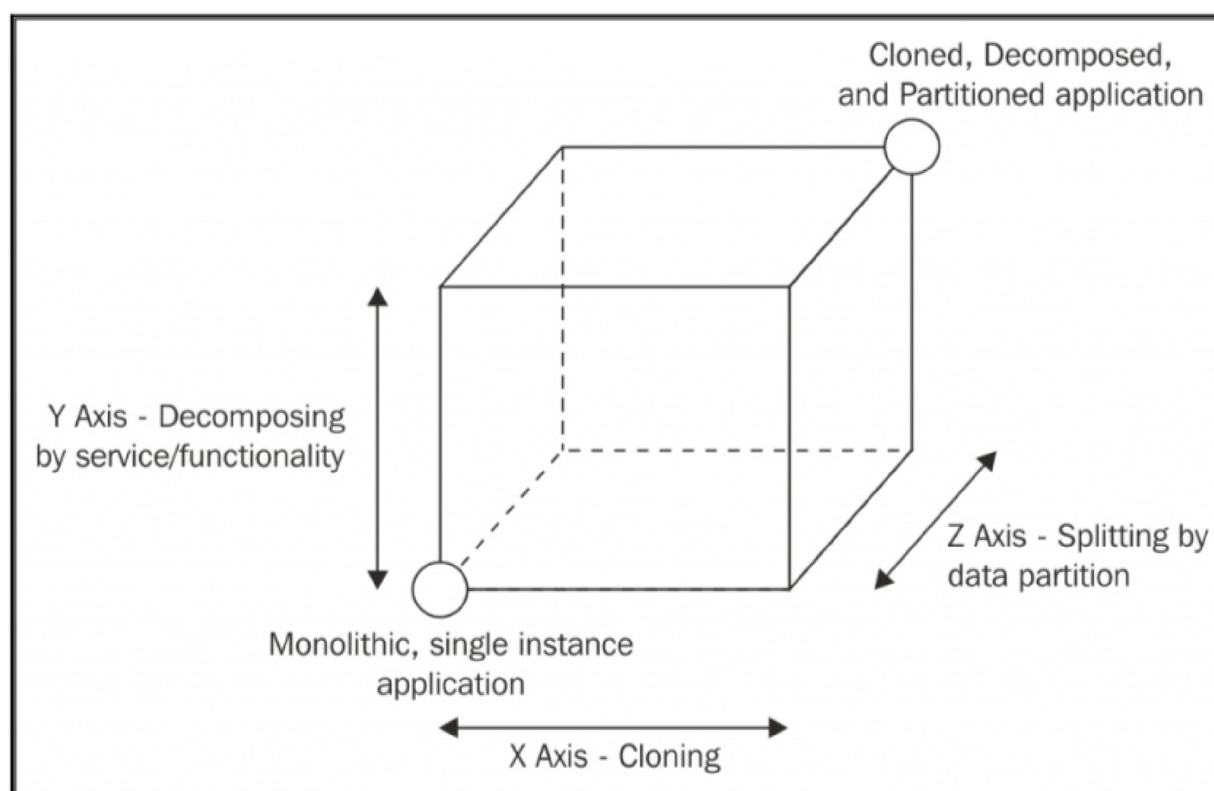
增长的一个障碍，但是通过纪律和精确的设计，我们可以把它变成一个优势。使用 `JavaScript` ，我们经常被迫使应用程序变得简单，并将其拆分成易于管理的部分，使其更易于扩展和分发。

可扩展性的三个维度

在谈到可伸缩性时，要理解的第一个基本原则是负载分布，这是将应用程序的负载分散到多个进程和机器上。有很多方法可以实现这一点， `Martin L. Abbott` 和 `Michael T. Fisher` 提出的“可扩展性的艺术”一书提出了一个巧妙的模型来表示它们，称为 `scale cube` 。该模型描述了以下三个方面的可扩展性：

- x轴：克隆，或者说复制
- y轴：按服务/功能分解
- z轴：按数据分区分割

这三个维度可以表示为一个立方体，如下图所示：



多维数据集的左下角表示应用程序在单个代码库（单片应用程序）中具有所有功能和服务，并在单个实例上运行。对于处理小型工作负载的应用程序或处于开发的早期阶段，这是一种常见的情况。

单片非缩放应用程序最直观的发展是沿着 `x` 轴向右移动，这很简单，大部分时间价格便宜（在开发成本方面）并且非常有效。这个技术背后的原理是基本的，就是克隆相同的应用程序 `n` 次，并让每个实例处理工作负载的 `1 / n` 。

沿 `y` 轴缩放意味着根据其功能，服务或用例来分解应用程序。在这种情况下，分解意味着创建不同的，独立的应用程序，每个应用程序都有其自己的代码库，有时还有自己的专用数据库，甚至是独立的UI。例如，常见的情况是将负责管理的应用程序的一部分与面向公众的产品分开。另一个例子是提取负责用户认证的服务，创建一个专用的认证服务器。按照功能划分应用程序的标准主要取决于其业务需求，用例，数据以及其他因素，我们将在本章后面介绍。有趣的是，这不仅是应用程序的体系结构，还是从开发的角度来看，它是最大的影响。正如我们将看到的，微服务是一个术语，目前通常与细粒度的 `y` 轴缩放关联。

最后一个缩放维度是 `z` 轴，应用程序以这样一种方式分割，即每个实例只负责整个数据的一部分。这是一种主要用于数据库的技术，也是水平分区或分片的名称。在此设置中，同一个应用程序有多个实例，每个实例都在数据的一个分区上运行，这是使用不同的标准确定的。例如，我们可以根据他们的国家（列表分区）或者基于他们姓氏的起始字母（范围分区）划分应用程序的用户，或者让一个散列函数决定每个用户所属的分区（散列分区）。然后可以将每个分区分配给我们应用程序的特定实例。使用数据分区需要在每个操作之前进行查找步骤，以确定应用程序的哪个实例负责给定的数据。正如我们所说的，数据分区通常在数据库级应用和处理，因为它的主要目的是克服处理大型单一数据集（磁盘空间有限，内存和网络容量有限）的问题。在应用程序级别应用它仅仅适用于复杂的分布式体系结构或非常特殊的用例，例如在构建依赖于数据持久性定制解决方案的应用程序，使用不支持分区的数据库时，或者在 `Google` 上构建应用程序时规模。考虑到其复杂性，只有在尺度立方体的 `x` 轴和 `y` 轴被充分利用之后，才应该考虑沿着 `z` 轴缩放应用程序。

在下一节中，我们将重点介绍两种最常用和最有效的技术来扩展 `Node.js` 应用程序，即通过功能/服务进行克隆和分解。

克隆和负载平衡

传统的多线程 `Web` 服务器通常只在分配给一台机器的资源不能再升级的时候才进行扩展，否则这个服务器的成本将高于简单地启动另一台机器的成本。 通过使用多个线程，传统的Web服务器可以利用服务器的所有处理能力，使用所有可用的处理器和内存。 但是，使用单个 `Node.js` 进程很难做到这一点，它是单线程的，在 `64` 位计算机上默认具有 `1.7 GB` 的内存限制（这需要增加一个名为 `--max_old_space_size` 的特殊命令行选项）。这意味着 `Node.js` 应用程序通常比传统的 `Web` 服务器更快地缩放，即使在单个机器的情况下，也能够利用其所有资源。

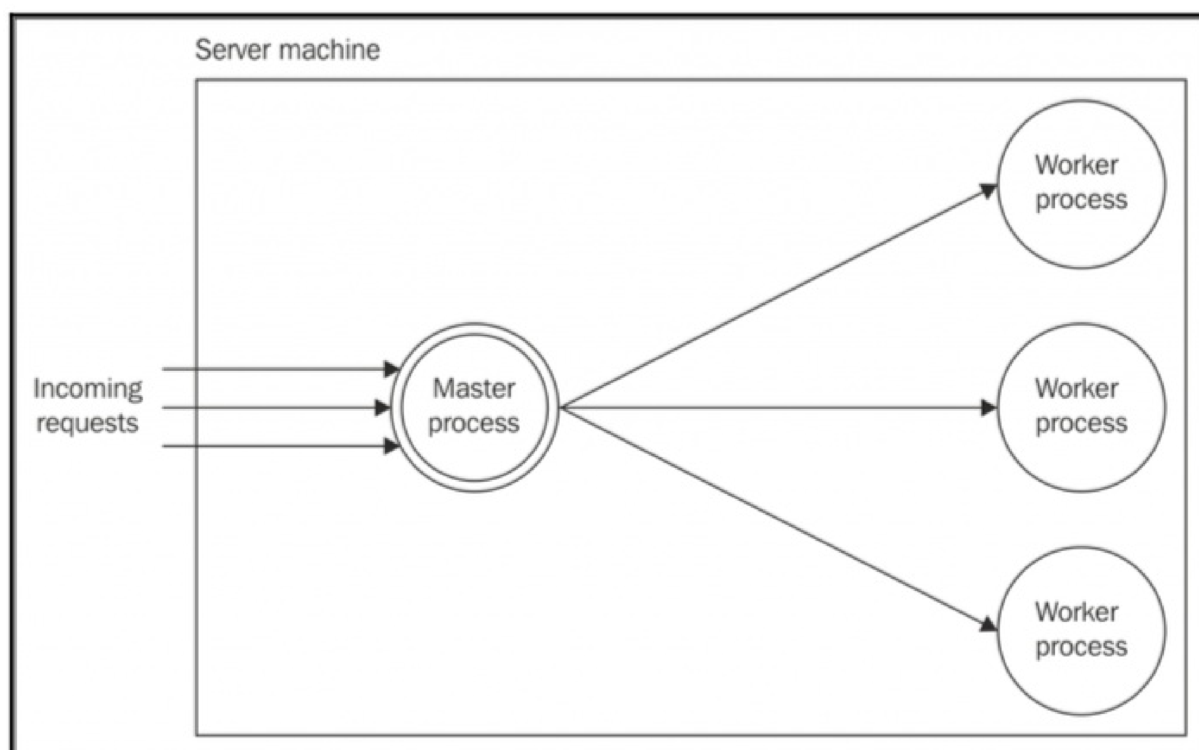
> 在 `Node.js` 中，垂直缩放（向单个机器添加更多资源）和水平缩放（将更多机器添加到基础架构）几乎是等价的概念; 事实上这两种技术类似，都是增加服务器的负载能力。

不要被愚弄，把这看作是一个缺点。相反，几乎被迫扩展对应用程序的其他属性，特别是可用性和容错性具有有益的影响。实际上，通过克隆来扩展 `Node.js` 应用程序相对比较简单，即使不需要获取更多的资源，也只是为了具有冗余的容错设置的目的而实现。 这也促使开发人员从应用程序的早期阶段考虑可伸缩性，确保应用程序不依赖任何不能在多个进程或机器间共享的资源。实际上，扩展应用程序的绝对先决条件是每个实例不必将通用信息存储在无法共享的资源（通常是硬件，如内

存或磁盘）上。例如，在 `Web` 服务器中，将会话数据存储在内存中或磁盘上是一种惯例，不适合缩放;相反，使用共享数据库将确保每个实例都可以访问相同的会话信息，无论它在哪里部署。 现在我们来介绍扩展 `Node.js` 应用程序的最基本机制：集群模块。

`cluster` 模块

在 `Node.js` 中，在单个机器上运行的不同实例之间分配应用程序负载的最简单模式是使用作为核心库一部分的 `cluster` 模块。 群集模块简化了相同应用程序的新实例的分叉，并自动将传入的连接分配到其中，如下图所示：



主进程负责产生大量进程（ `worker` ），每个进程代表我们想要扩展的应用程序的一个实例。每个传入连接然后分布在克隆的 `worker` ，分散在他们的负载。

关于 `cluster` 模块行为的注意事项

在 `Node.js 0.8` 和 `0.10` 中， `cluster` 模块在工作人员之间共享相同的服务器套接字，并离开操作系统，负载平衡跨可用工作者的传入连接。但是，这种方法存在问题。实际上，操作系统用于在工作人员之间分配负载的算法并不意味着对网络请求进行负载平衡，而是调度进程的执行。因此，在所有情况下，分配并不总是一致的；往往只有一小部分工人获得了大部分的工作量。这种类型的行为对于操作系统调度程序是有意义的，因为它着重于最小化不同进程之间的上下文切换。简而言之， `cluster` 模块在 `Node.js <= 0.10` 中不能充分发挥其潜力。 但是，情况从版本 `0.11.2` 开始变化，在主进程中包含明确的循环负载平衡算法，这确保请求在

所有工作者中均匀分布。新的负载均衡算法默认情况下在 `Windows` 以外的所有平台上启用，可以通过设置变量 `cluster.schedulingPolicy` ，使用常量 `cluster.SCHED_RR` （循环）或 `cluster.SCHED_NONE` （由操作系统处理）。

> 轮循算法轮流在可用服务器上均匀分配负载。第一个请求被转发到第一个服务器，第二个请求转发到列表中的下一个服务器，依此类推。 当列表结束时，迭代从头开始。 这是最简单和最常用的负载均衡算法之一；然而，这不是唯一的一个。 更复杂的算法允许分配优先级，选择负载最少的服务器或响应时间最快的服务器。 您可以在这两个 `Node.js` 问题中找到关于集群模块演变的更多细节： https://github.com/nodejs/node-v0.x-archive/issues/4435 和 https://github.com/nodejs/node-v0.x-archive/issues/3241

建立一个简单的HTTP服务器

现在开始研究一个例子。 让我们构建一个小型的 `HTTP` 服务器，使用集群模块进行克隆和负载平衡。 首先，我们需要一个应用程序来扩展；对于这个例子我们不需要太多，只是一个非常基本的 `HTTP` 服务器。

我们创建一个名为 `app.js` 的文件，其中包含以下代码：

```
const http = require('http');
const pid = process.pid;
http.createServer((req, res) => {
  for (let i = 1e7; i > 0; i--) {}
  console.log(`Handling request from ${pid}`);
  res.end(`Hello from ${pid}\n`);
}).listen(8080, () => {
  console.log(`Started ${pid}`);
});
```

我们刚刚构建的 `HTTP` 服务器通过发回包含 `PID` 的消息来响应任何请求; 这将有助于识别哪个应用程序实例正在处理请求。另外，为了模拟一些实际的 `CPU` 工作，我们执行一个空循环 `1000` 万次；没有这个，考虑到我们要为这个例子运行的小规模的测试，服务器负载几乎是没有的。

> 我们想扩展的 `app` 模块可以是任何东西，也可以使用Web框架来实现，例如 `Express` 。

现在，我们可以像往常一样运行应用程序，并使用浏览器或 `curl` 向 `http://localhost:8080` 发送请求，检查是否所有程序都按预期工作。

我们也可以尝试测量服务器每秒只能使用一个进程处理的请求；为此，我们可以使用网络基准测试工具，如siege或Apache ab：

```
siege -c200 -t10S http://localhost:8080
```

用 `ab` ，命令行会非常相似：

```
ab -c200 -t10 http://localhost:8080/
```

上述命令将以 `200` 个并发连接加载服务器 `10` 秒钟。 作为参考，具有 `4` 个处理器的系统的结果是每秒 `90` 个事务的顺序，平均 `CPU` 利用率仅为 `20%` 。

> 请记住，我们将在本章中执行的负载测试故意做成最简单和最小的，仅供参考和学习之用。他们的结果不能提供我们正在分析的各种技术的性能的 `100%` 准确的评估。



使用 `cluster` 模块进行扩展

现在让我们尝试使用集群模块来扩展我们的应用程序。 我们来创建一个名为 `clusteredApp.js` 的新模块：

```
const cluster = require('cluster');
const os = require('os');

if(cluster.isMaster) {
  const cpus = os.cpus().length;
  for (let i = 0; i < cpus; i++) {  // [1]
    cluster.fork();
  }
} else {
  require('./app');  // [2]
}
```

正如我们所看到的，使用 `cluster` 模块只需要很少的努力。我们来分析一下发生的事情：

- 当我们从命令行启动 `clusteredApp` 时，我们实际上正在执行主进程。 `cluster.isMaster` 变量设置为 `true` ，我们需要做的唯一工作是使用 `cluster.fork()` 来 `fork` 当前进程。 在前面的示例中，我们启动的系统中的CPU数量与可用的所有处理能力相同。
- 当从主进程执行 `cluster.fork()` 时，当前主模块（ `clusteredApp` ）再次

运行，但是这次是工作模式（ `cluster.isWorker` 设置为 `true` ，而 `cluster.isMaster` 为 `false` ）。当应用程序作为 `worker` 运行时，它可以开始做一些实际的工作。 在我们的例子中，我们加载了 `app` 模块，它实际上启动了一个新的 `HTTP` 服务器。

> 记住每个 `worker` 都是一个不同的 `Node.js` 进程，它有自己的事件循环，内存空间和加载的模块。

有趣的是，注意到集群模块的使用基于循环模式，这使得运行多个应用程序的实例变得非常简单：

```
if (cluster.isMaster) {
  // fork()
} else {
  // do work
}
```

在底层，集群模块使用了 `child_process.fork() API` （我们已经在 `Chapter 9, Advanced Asynchronous Recipes` 中已经遇到了这个 `API` ），因此我们也在 `master` 和 `worker` 之间有一个可用的通信通道。 工人的实例可以通过变量 `cluster.workers` 访问，所以向所有人发送消息就像运行下面几行代码一样简单：

```
Object.keys(cluster.workers).forEach(id => {
  cluster.workers[id].send('Hello from the master');
});
```

现在，让我们尝试以集群模式运行我们的 `HTTP` 服务器。 我们可以像往常一样启动 `clusteredApp` 模块来做到这一点：

```
node clusteredApp
```

如果我们的机器有多个处理器，我们应该看到一些 `worker` 正在被主进程一个接一个地启动。例如，在一个有四个处理器的系统中，终端应该是这样的：

```
Started 14107
Started 14099
Started 14102
Started 14101
```

如果我们现在尝试使用 `URL http://localhost：8080` 再次访问我们的服务器，我们应该注意到每个请求都会返回一个带有不同 `PID` 的消息，这意味着这些请求已经由不同的 `worker` 处理，确认 负载正在其中分配。

现在我们可以尝试再次加载测试我们的服务器：

```
siege -c200 -t10S http://localhost:8080
```

这样，我们就能够发现通过在多个进程中扩展应用程序所获得的性能提升。 作为参考，通过在具有 `4` 个处理器的 `Linux` 系统中使用 `Node.js 6` ，在平均 `CPU` 负载为 `90%` 的情况下，性能提高应该是 `3` 倍（为 `270 trans / sec` ，比起 `90 trans / sec` ）。

`cluster` 模块的可拓展性和可用性

正如我们已经提到的那样，扩展应用程序还带来了其他优点，特别是即使在出现故障或崩溃时也能保持一定的服务水平的能力。 这个属性也被称为弹性，它有助于系统的可用性。

通过启动同一应用程序的多个实例，我们正在创建一个冗余系统，这意味着如果一个实例由于某种原因而关闭，我们仍然有其他实例可以为请求提供服务。 这种模式使用集群模块非常简单。 让我们看看它是如何工作的！

我们以上一节的代码为起点。特别是，我们修改 `app.js` 模块，使其在随机时间间隔后崩溃：

```
// 在app.js的最后
setTimeout(() => {
  throw new Error('Ooops');
}, Math.ceil(Math.random() * 3) * 1000);
```

在这种变化的情况下，我们的服务器在 `1` 到 `3` 之间的随机数字时间之后退出，出现错误。在真实的情况下，这会导致我们的应用程序停止工作，当然，服务请求，除非我们使用一些外部工具来监视其状态并自动重启。但是，如果我们只有一个实例，那么由应用程序的启动时间引起的重新启动之间可能会有一个不可忽略的延迟。 这意味着在这些重新启动期间，应用程序不可用。拥有多个实例会确保我们总是有一个备份系统来处理即将到来的请求，即使其中一个工作者失败。

使用 `cluster` 模块，只要我们检测到一个错误代码被终止，我们所要做的就是产生一个新的 `worker` 。 那么我们来修改 `clusteredApp.js` 模块来考虑这个问题：

```
if (cluster.isMaster) {
  // ...
  cluster.on('exit', (worker, code) => {
    if (code != 0 && !worker.suicide) {
      console.log('Worker crashed. Starting a new worker');
      cluster.fork();
    }
  });
} else {
  require('./app');
}
```

在前面的代码中，一旦主进程收到 `exit` 事件，我们检查进程是有意终止的还是错误的结果；我们通过检查状态码和 `worker.exitedAfterDisconnect` 来实现这一点，这表明工作者是否被明确地终止了。 如果我们确认过程因错误而终止，我们启动一个新的 `worker` 。有意思的是，当崩溃的 `worker` 重新启动时，其他 `worker` 仍然可以提供请求，从而不会影响应用程序的可用性。

为了测试这个假设，我们可以试着用 `siege` 再次重启我们的服务器。当压力测试完成时，我们注意到 `siege` 产生的各种指标中还有一个衡量应用程序可用性的指标。 预期的结果会是这样的：

```
Transactions: 3027 hits
Availability: 99.31%
Failed transactions: 21
```

```
..10/01_cluster
Transactions: 2895 hits
Availability: 95.68%
Failed transactions: 32
➜ 01_cluster git:(master) ✗
```

请记住，这个结果可能会有很大的变化。它在很大程度上取决于正在运行的实例的数量以及它们在测试期间崩溃的次数，但是它应该很好地指出我们的解决方案是如何工作的。 前面的数字告诉我们，尽管我们的应用程序不断崩溃，但是在超过了 `3027` 次请求中只有 `21` 次失败的请求。 在我们构建的示例场景中，大部分失败的请求将由崩溃期间已建立连接的中断引起。

事实上，当发生这种情况时， `siege` 将会打印出如下错误：

```
[error] socket: read error Connection reset by peer sock.c:479:
Connection reset by peer
```

不幸的是，为了防止这类类型的错误，我们能够做的不多，特别是当应用程序因崩溃而终止时。尽管如此，我们的解决方案证明是可行的，对于经常崩溃的应用程序，使用 `cluster` ，其可拓展性性并不差。

零宕机重启

当代码需要更新时， `Node.js` 应用程序也可能需要重新启动。因此，在这种情况下，拥有多个实例可以帮助维护我们应用程序的可用性。 当我们不得不故意重新启动一个应用程序来更新它时，会出现一个小窗口，在这个窗口中应用程序将重新启动并且无法为请求提供服务。如果我们正在更新我们的个人博客，这是可以接受的，但对于具有服务水平协议（ `SLA` ）的专业应用程序就不行了，或者作为持续交付过程的一部分经常更新的专业应用程序。解决方案是实现零宕机重新启动，更新应用程序的代码而不影响其可用性。

使用 `cluster` 模块，这又是一项非常简单的任务；该模式包括一次重启一个 `worker` 。这样，剩余的 `worker` 可以继续操作和维护可用应用程序的服务。

然后，让我们将这个新模块添加到我们的集群服务器；我们所要做的就是添加一些由主进程执行的新代码（看 `clusteredApp.js` 文件）：

```
const cluster = require('cluster');
const os = require('os');

if (cluster.isMaster) {
  const cpus = os.cpus().length;
  for (let i = 0; i < cpus; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code) => {
    if (code != 0 && !worker.exitedAfterDisconnect) {
      console.log('Worker crashed. Starting a new worker');
      cluster.fork();
    }
  });

  process.on('SIGUSR2', () => {
    console.log('Restarting workers');
    const workers = Object.keys(cluster.workers);

    function restartWorker(i) {
      if (i >= workers.length) return;
      const worker = cluster.workers[workers[i]];
      console.log(`Stopping worker: ${worker.process.pid}`);
      worker.disconnect();

      worker.on('exit', () => {
        if (!worker.suicide) return;
        const newWorker = cluster.fork();
        newWorker.on('listening', () => {
          restartWorker(i + 1);
        });
      });
    }
    restartWorker(0);
  });
} else {
  require('./app');
}
```

这是前面的代码的工作原理：

1. 一旦接收到 `SIGUSR2` 信号，则触发 `worker` 重新启动。
2. 我们定义一个名为 `restartWorker()` 的迭代器函数。异步迭代 `cluster.workers` 的每一项。
3. `restartWorker()` 函数的第一个任务是通过调用 `worker.disconnect()` 来优雅地停止工作。
4. 当终止的进程退出时，我们可以产生一个新的 `worker` 。
5. 只有当新的 `worker` 准备好并且正在侦听新的连接时，我们才可以通过调用迭代的下一步来重新启动下一个 `worker` 。

> 由于我们的程序使用了 `UNIX` 信号，因此在 `Windows` 系统上无法正常工作（除非您在 `Windows 10` 中使用最新的 `Windows` 子系统）。信号是实现我们的解决方案的最简单的机制。但是，这不是唯一的；实际上，其他方法包括侦听来自套接字，管道或标准输入的命令。

现在我们可以通过运行 `clusteredApp` 模块然后发送一个 `SIGUSR2` 信号来测试我们的零宕机重启。但是，首先我们需要获取主进程的 `PID` ；以下命令可用于从所有正在运行的进程的列表中识别它：

```
ps af
```

主进程应该是一组节点进程的父节点。一旦我们有我们正在寻找的 `PID` ，我们可以发送信号给它：

```
kill -SIGUSR2 <PID>
```

现在， `clusteredApp` 应用程序的输出应该显示如下所示：

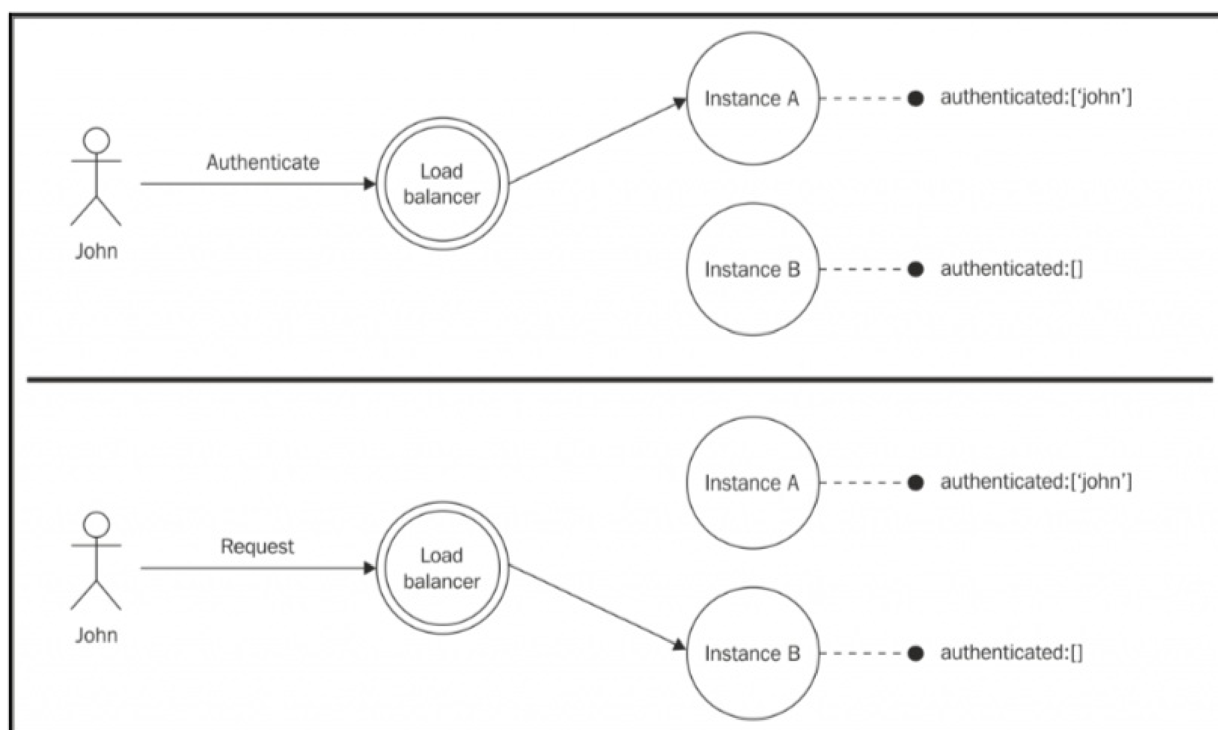
```
Restarting workers
Stopping worker: 19389
Started 19407
Stopping worker: 19390
Started 19409
```

我们可以尝试再次使用 `siege` 来验证我们在重新启动 `worker` 时对应用程序的可用性没有太大的影响。

> pm2是一个基于 `cluster` 的小型实用程序，它提供负载平衡，过程监控，零宕机重启等功能。

处理有状态的通信

`cluster` 模块不适用于有状态通信，应用程序维护的状态在各个实例之间不共享。这是因为属于相同有状态会话的不同请求可能会由应用程序的不同实例处理。这不是一个仅限于 `cluster` 模块的问题，但通常它适用于任何种类的无状态负载均衡算法。例如，考虑下图所描述的情况：

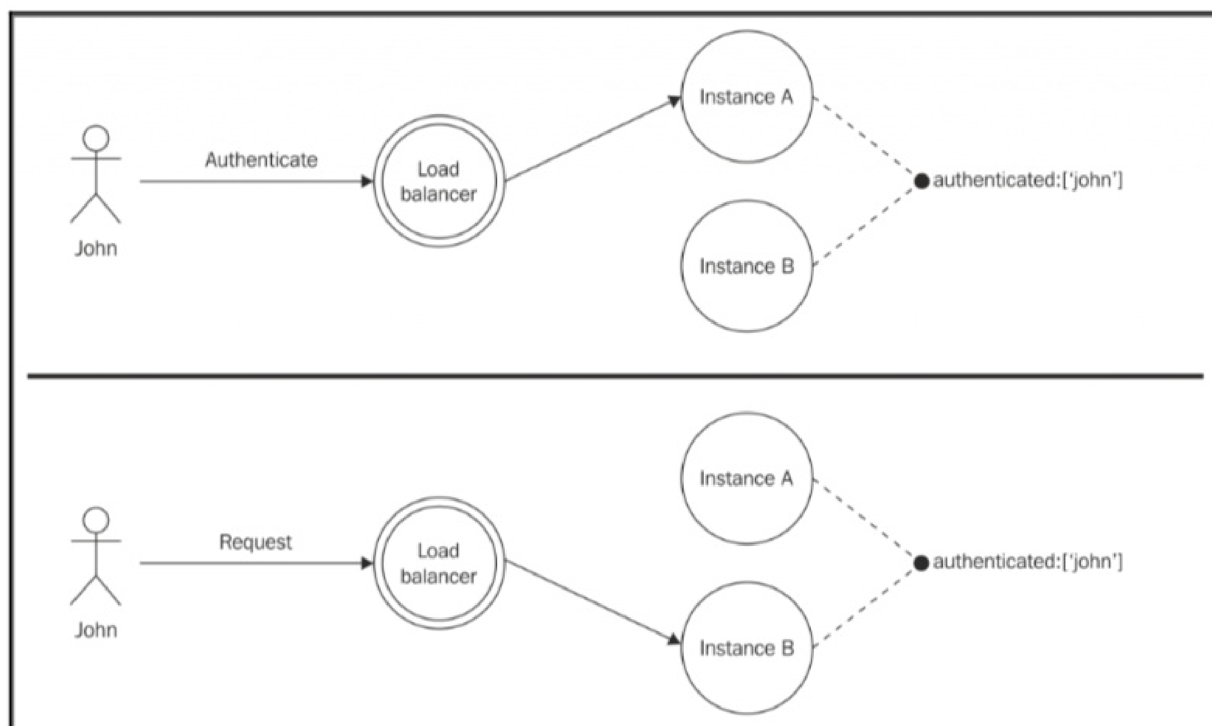


用户John最初发送一个请求到我们的应用程序来验证自己身份，但是操作的结果是在本地注册的（例如在内存中），所以只有接收到认证请求的应用程序实例（实例 `A` ）知道 `John` 已成功通过身份验证。当 `John` 发送一个新的请求时，负载平衡器可能会将它转发给应用程序的另一个实例，实际上它不具有 `John` 的认证细节，因此拒绝执行该操作。我们刚刚描述的应用程序不能按比例缩放，但幸运的是，我们可以通过两个简单的解决方案来解决问题。

跨多个实例共享状态

要实现在所有实例之间共享状态，我们必须使用有状态通信来扩展应用程序。这可以通过共享数据存储容易地实现，例如像PostgreSQL，MongoDB或CouchDB，或者甚至更好，我们可以使用内存存储，如Redis或Memcached。

下图概述了这个简单有效的解决方案：

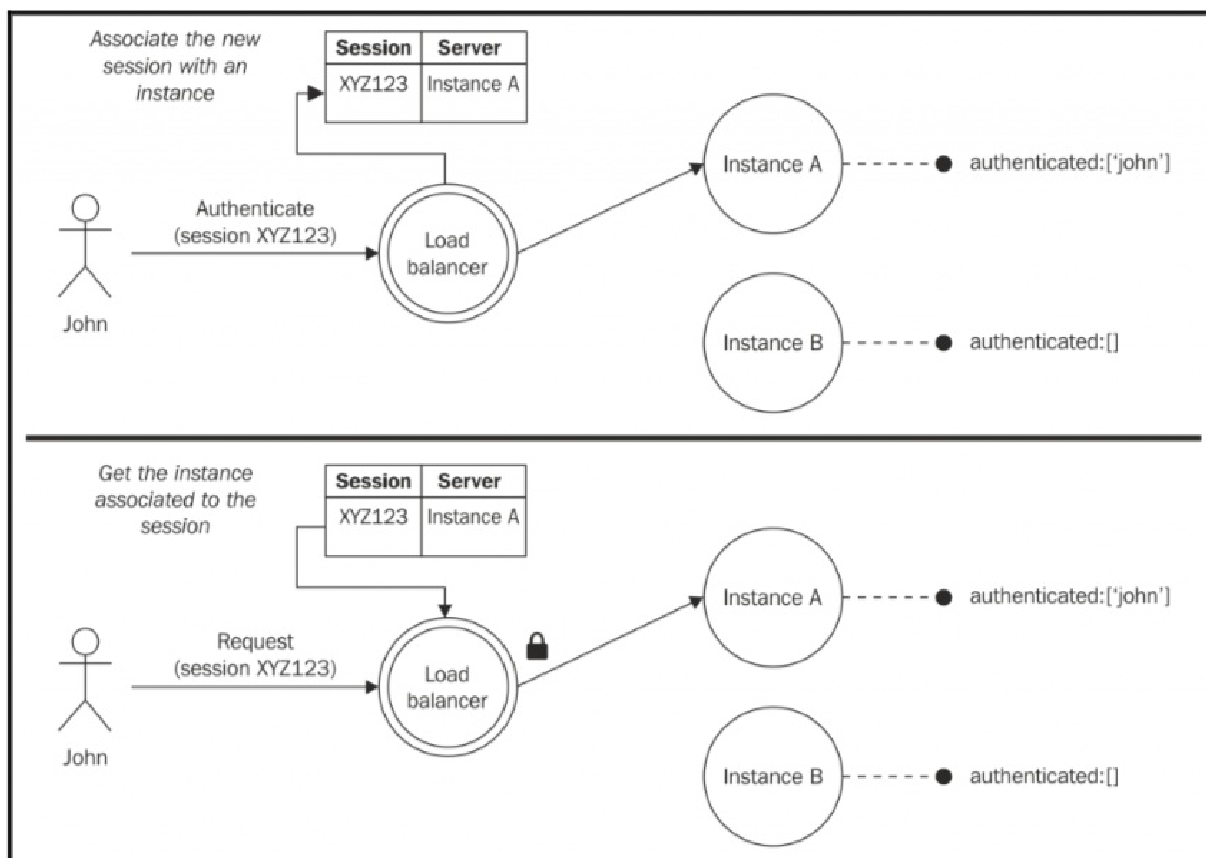


在通信状态中使用共享存储的唯一缺点是，这并不总是容易实现的，例如，我们可能会使用现有的库在内存中保持通信状态；无论如何，如果我们有一个现有的应用程序，那要在现有应用程序上增加共享数据存储则需要更改应用程序的代码（如果它尚未支持）。正如我们接下来会看到的那样，看接下来这个解决方案。

粘性负载均衡

我们必须支持有状态通信的另一种方法是使负载均衡器始终将与会话相关的所有请求都路由到应用程序的同一实例。这种技术也被称为粘性负载均衡。

下图说明了涉及此技术的简化方案：



从上图可以看出，当负载均衡器接收到与 `session` 相关的请求时，它会创建一个映射，其中包含由负载平衡算法选择的一个特定实例。负载平衡器下一次接收到来自同一个会话的请求时，会绕过负载平衡算法，选择之前与会话关联的应用程序实例。我们刚刚描述的特定技术涉及检查与请求相关的 `session ID` （通常由应用程序或负载平衡器本身包含在 `cookie` 中）。

将有状态连接关联到单个服务器的更简单的替代方法是记住执行请求的客户端的 `IP` 地址。通常，将 `IP` 提供给一个 `hash` 函数，该函数生成一个代表指定接收请求的应用程序实例的 `ID` 。这种技术的优点是不需要负载均衡器记住关联。但是，对于频繁更换IP的设备，例如在不同网络上漫游时，它不起作用。

> `cluster` 模块默认不支持粘性负载均衡；不过，它可以添加一个名为sticky-session的 `npm` 库来实现这一点。

粘性负载均衡的一个大问题是，它使得拥有冗余系统的大部分优点失效，其中应用程序的所有实例都是相同的，并且实例可以最终替代另一个停止工作的实例。出于这些原因，建议避免在共享存储中维护任何会话状态使用粘性负载均衡，在根本不需要有状态通信的应用程序（例如，通过在请求中包含状态）使用粘性负载均衡。

> 对于需要粘性负载平衡的库的一个真实例子，可以看看socket.io

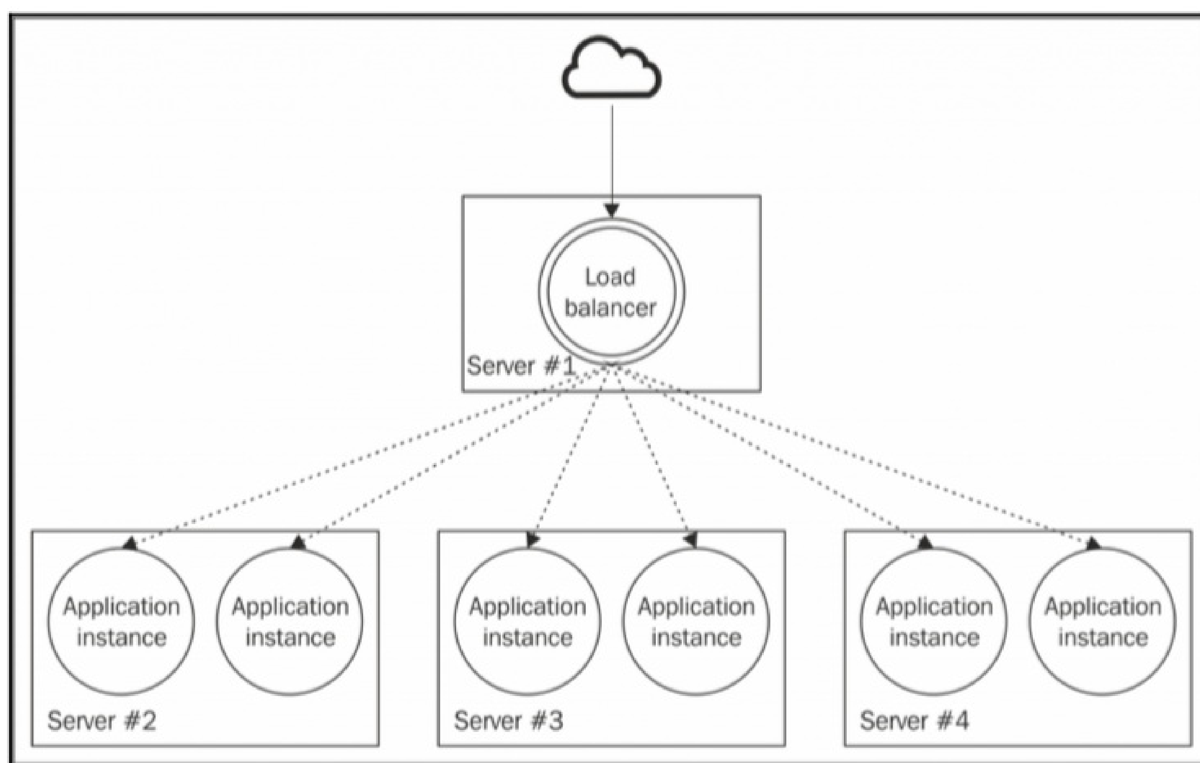
使用反向代理进行缩放

`cluster` 模块不是我们必须扩展 `Node.js Web` 应用程序的唯一选项。事实上，更多的传统技术往往是首选，因为它们在生产环境中更易于使用。

替代 `cluster` 的另一种方法是启动在不同端口或计算机上运行的同一应用程序的多个独立实例，然后使用反向代理（或网关）提供对这些实例的访问权限，从而将流量分配到这些实例。在这个配置中，我们没有一个主进程将请求分发给一组工作者，而是在同一台机器上运行的一组不同的进程（使用不同的端口），或者分散在网络内的不同机器上。为了向我们的应用程序提供单一的访问点，我们可以使用一个反向代理，放置在客户端和应用程序的实例之间的一个特殊的设备或服务，它接受任何请求并将其转发到目标服务器，并将结果返回给 客户端，而这些对客户端来说都是透明的。在这种情况下，反向代理也用作负载平衡器，将请求分发到应用程序的实例中。

> 有关反向代理和转发代理之间差异的明确说明，可以参阅Apache HTTP服务器文档

下图显示了一个典型的多进程多机配置，其中一个反向代理充当负载均衡器的前端：



对于Node.js应用程序，选择此方法取代 `cluster` 模块的原因有很多：

- 反向代理可以将负载分布到多个机器上，而不仅仅是几个进程；
- 市场上最流行的反向代理支持粘性负载均衡；
- 反向代理可以将请求路由到任何可用的服务器，而不管其编程语言或平台；
- 我们可以选择更强大的负载均衡算法；
- 许多反向代理还提供其他服务，例如URL重写，缓存，SSL终止点，甚至可以使用的完全成熟的Web服务器的功能，例如，为静态文件提供服务。

也就是说，如果需要， `cluster` 模块也可以很容易地与反向代理结合使用；例如，使用 `cluster` 在单个机器内部垂直缩放，然后使用反向代理在不同节点之间水平缩放。

> 模式：使用反向代理来平衡在不同端口或机器上运行的多个实例之间的应用程序负载。

对于反向代理实现负载均衡器，我们有很多选择；一些流行的解决方案如下：

- Nginx：这是一个基于 `非阻塞I/O` 模型的 `Web` 服务器，反向代理和负载均衡器。
- HAProxy：这是一个用于 `TCP/HTTP` 流量的快速负载均衡器。
- 基于 `Node.js` 的代理：有很多解决方案可以直接在 `Node.js` 中实现反向代理和负载均衡器。这可能有优点和缺点，我们将在后面看到。
- 基于云的代理服务器：在云计算时代，利用负载均衡器作为服务并不罕见。这可能很方便，因为它基本不需要维护，通常具有高度的可扩展性，有时它可以支持动态配置以实现按需扩展。

在本章接下来的几节中，我们将分析一个使用 `Nginx` 的配置示例，接下来我们还将使用 `Node.js` 来构建我们自己的负载均衡器。

使用 `Nginx` 进行负载平衡

为了说明专用反向代理如何工作，我们现在将构建基于Nginx的可扩展架构，但首先我们需要安装它。 我们可以按照 http://nginx.org/en/docs/install.html 上的说明来做到这一点。

在最新的 `Ubuntu` 系统上，您可以使用以下命令快速安装 `Nginx` ：

```
sudo apt-get install nginx
```

在 `Mac OSX` 上，您可以使用brew：

```
brew install nginx
```

由于我们不打算使用 `cluster` 来启动服务器的多个实例，因此我们需要稍微修改应用程序的代码，以便我们可以使用命令行参数指定侦听端口。这将允许我们在不同的端口上启动多个实例。我们再来考虑我们的示例应用程序（ `app.js` ）的主要模块：

```
const http = require('http');
const pid = process.pid;

http.createServer((req, res) => {
  for (let i = 1e7; i > 0; i--) {}
  console.log(`Handling request from ${pid}`);
  res.end(`Hello from ${pid}\n`);
}).listen(process.env.PORT || process.argv[2] || 8080, () => {
  console.log(`Started ${pid}`);
});
```

另一个不使用 `cluster` 的原因是其在发生崩溃时无法自动重启。幸运的是，这很容易通过使用专用的管理程序来解决，该管理程序监视我们的应用程序并在必要时重新启动的外部进程。可能的选择如下：

- 基于 `Node.js` 的 `supervisors` ，如forever或pm2
- 基于 `OS` 的 `supervisors` ，例如upstart，systemd或者runit
- 更高级的 `supervisors` 解决方案，如monit或supervisor。

对于这个例子，我们将使用 `forever` ，这是我们使用最简单，最直接的。 我们可以通过运行以下命令来全局安装它：

```
npm install forever -g
```

下一步是启动我们的应用程序的四个实例，全部在不同的端口上，使用 `forever` ：

```
forever start app.js 8081
forever start app.js 8082
forever start app.js 8083
forever start app.js 8084
```

我们可以使用以下命令检查已启动进程的列表：

```
forever list
```

现在需要将 `Nginx` 服务器配置为负载平衡器。

首先，我们需要根据你的系统来确定 `nginx.conf` 文件的位置。一般是在 `/usr/local/nginx/conf` ， `/etc/nginx` ，或者 `/usr/local/etc/nginx` 。

接下来，我们打开 `nginx.conf` 文件并应用以下配置，这是获得实现负载均衡所需的最基础的配置：

```
http {
  # [...]

  upstream nodejs_design_patterns_app {
    server 127.0.0.1:8081;
    server 127.0.0.1:8082;
    server 127.0.0.1:8083;
    server 127.0.0.1:8084;
  }

  # [...]

  server {
      listen       80;

      location / {
        proxy_pass         http://nodejs_design_patterns_app;
      }
  }

  # [...]
}
```

对于配置文件，基本不用解释。在 `upstream nodejs_design_patterns_app` 部分，我们定义了用于处理网络请求的后端服务器列表，然后在 `server` 部分中指定了 `proxy_pass` 指令，这本质上告诉 `Nginx` 将任何请求转发给我们之前定义的服务器组（ `nodejs_design_patterns_app` ）。就是这样，现在我们只需要用以下命令重新加载 `Nginx` 配置：

```
nginx -s reload
```

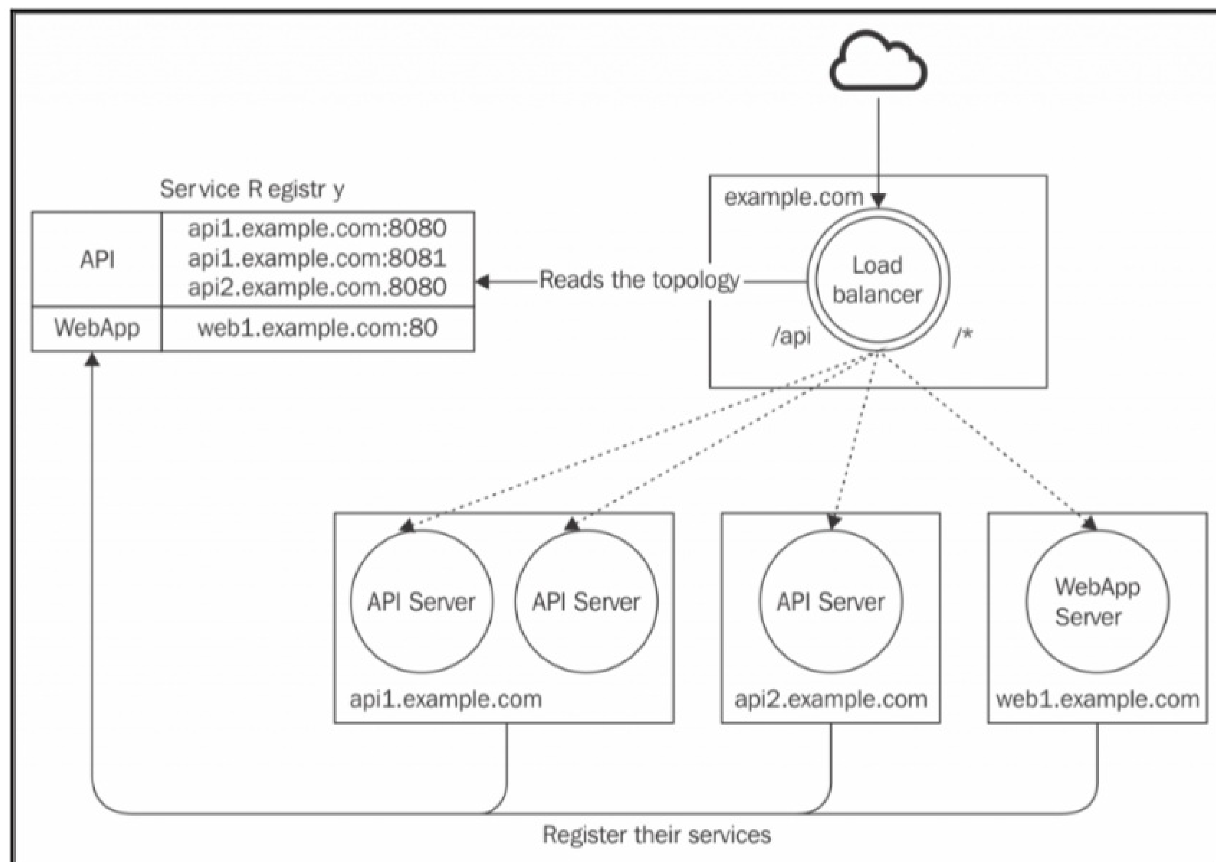
我们的系统现在应该已经启动并且正在运行，已经准备好接受请求并且平衡 `Node.js` 应用程序的四个实例的流量。 只需在您的浏览器打开地址 http://localhost，查看我们的`Nginx`服务器如何平衡流量。

使用服务注册表

现代基于云的基础架构的一个重要优势是能够基于当前的运行情况，预测的流量动态调整应用的容量；这也被称为动态缩放。如果实施得当，这种做法可以极大地降低 `IT` 基础架构的成本，同时保持应用程序的高可用性和响应能力。

这个想法很简单：如果我们的应用程序正在经历由流量高峰造成的性能下降，我们会自动产生新的服务器来应对增加的负载。我们也可以决定在某些时间关闭一些服务器，例如晚上，当我们知道流量将会减少时，在早上再次重新启动它们。该机制要求负载均衡器随时了解当前的网络拓扑结构，随时了解哪台服务器处于运行状态。

解决此问题的常见模式是使用称为服务注册中心的中央存储库，该中心存储库跟踪正在运行的服务器及其提供的服务。下图显示了前端具有负载平衡器的多服务架构，使用服务注册表进行动态配置：



上述架构假定存在两个服务 `API` 和 `WebApp` 。负载均衡器将到达 `/ api` 节点的请求分发给实现 `API` 服务的所有服务器，而其余请求分布在实现WebApp服务的服务器上。负载均衡器获取使用服务注册表的服务器列表。

为了使其完全自动化运行，每个应用程序实例在联机时必须自己注册到服务注册表，并在其停止时取消注册。通过这种方式，负载均衡器可以始终拥有最新的服务器视图和网络上可用的服务。

> 模式（服务注册表）：使用中央资源库来存储和管理服务器的最新视图以及系统中可用的服务。

这种模式不仅可以应用于负载平衡，还可以更普遍地作为从提供服务的服务器分离服务类型的一种方式。我们可以将其视为适用于网络服务的服务定位器的设计模式。

使用 `http-proxy` 和 `Consul` 实现动态负载均衡器

为了实现粘性负载均衡，我们可以使用反向代理，例如 `Nginx` 或 `HAProxy` ；我们所需要做的就是使用自动服务更新其配置，然后强制负载均衡器选择更改。 对于 `Nginx` ，可以使用以下命令行完成：

```
nginx -s reload
```

使用基于云的解决方案可以获得相同的结果，但我们有第三种更熟悉的替代方案，可以使用我们最喜欢的平台。

我们都知道 `Node.js` 是构建任何网络应用程序的好工具；正如我们所说，这正是其主要设计目标之一。那么，为什么不建立一个只使用 `Node.js` 的负载均衡器呢？ 这将给我们更多的自由，并允许我们直接在我们的定制负载平衡器中实现任何类型的模式或算法，包括我们现在要探索的负载平衡器，使用服务注册表的动态负载平衡。在这个例子中，我们将使用Consul作为服务注册表。

在这个例子中，我们想要复制我们在上一节中看到的多服务体系结构，为此，我们将主要使用三个 `npm` 包：

- http-proxy：这是一个库，用于简化 `Node.js` 中代理和负载均衡器的创建
- portfinder：这是一个允许发现系统中的自由端口的库
- consul：这是一个图书馆，允许服务在 `consul` 登记

让我们开始实施我们的服务。 它们是简单的 `HTTP` 服务器，就像我们迄今用来测试 `cluster` 和 `Nginx` 的 `HTTP` 服务器一样，但是这次我们希望每个服务器都在服务注册表启动的时候注册自己。

让我们看看这看起来如何（文件 `app.js` ）：

```
const http = require('http');
const pid = process.pid;
const consul = require('consul')();
const portfinder = require('portfinder');
const serviceType = process.argv[2];

portfinder.getPort((err, port) => {
  const serviceId = serviceType+port;
  consul.agent.service.register({
    id: serviceId,
    name: serviceType,
    address: 'localhost',
    port: port,
    tags: [serviceType]
  }, () => {

    const unregisterService = (err) => {
      consul.agent.service.deregister(serviceId, () => {
        process.exit(err ? 1 : 0);
      });
    };

    process.on('exit', unregisterService);
    process.on('SIGINT', unregisterService);
    process.on('uncaughtException', unregisterService);

    http.createServer((req, res) => {
      for (let i = 1e7; i > 0; i--) {}
      console.log(`Handling request from ${pid}`);
      res.end(`${serviceType} response from ${pid}\n`);
    }).listen(port, () => {
      console.log(`Started ${serviceType} (${pid}) on port ${por
t}`);
    });
  });
});
```

在前面的代码中，有一些部分值得我们关注：

- 首先，我们使用 `portfinder.getPort` 来发现系统中的一个空闲端口（默认情况下， `portfinder` 从 8000 端口开始搜索）。
- 接下来，我们使用 `Consul` 库在注册表中注册一项新服务。服务定义需要几个属性： `id` （服务的唯一名称）， `name` （标识服务的通用名称）， `address` 和 `port` （用于标识如何访问服务）， `tags` （可选的标签数组用于过滤和分组服务）。我们使用 `serviceType` （我们将其作为命令行参数）来指定服务名称并添加标签。这将允许我们识别集群中可用的相同类型的所有服务。
- 此时我们定义了一个名为 `unregisterService` 的函数，它允许我们在集群中定义相同类型的服务。
- 我们使用 `unregisterService` 作为清理函数，以便程序运行时关闭（无论是

人为关闭还是意外关闭），从取消注册。
- 最后，我们为 `portfinder` 发现的端口上的服务启动 `HTTP` 服务器。

现在是实施负载均衡器的时候了。我们通过创建一个名为 `loadBalancer.js` 的新模块来实现这一点。首先，我们需要定义一个路由表来将 `URL` 路径映射到服务：

```
const routing = [{
  path: '/api',
  service: 'api-service',
  index: 0
}, {
  path: '/',
  service: 'webapp-service',
  index: 0
}];
```

`routing` 数组中的每个项目都包含用于处理到达映射路径的请求的服务。`index` 属性将用于循环给定服务的请求。

让我们通过实现 `loadbalancer.js` 的第二部分来看看它是如何工作的：

```
const proxy = httpProxy.createProxyServer({});
http.createServer((req, res) => {
  let route;
  routing.some(entry => {
    route = entry;
    //Starts with the route path?
    return req.url.indexOf(route.path) === 0;
  });

  consul.agent.service.list((err, services) => {
    const servers = [];
    Object.keys(services).filter(id => {
      if (services[id].Tags.indexOf(route.service) > -1) {
        servers.push(`http://${services[id].Address}:${services[
id].Port}`)
      }
    });

    if (!servers.length) {
      res.writeHead(502);
      return res.end('Bad gateway');
    }

    route.index = (route.index + 1) % servers.length;
    proxy.web(req, res, {target: servers[route.index]});
  });
}).listen(8080, () => console.log('Load balancer started on port
 8080'));
```

这就是我们如何实现基于 `Node.js` 的负载均衡器：

1. 首先，我们需要 `consul` ，以便我们可以访问注册表。接下来，我们实例化一个 `http-proxy` 对象并启动一个普通的 `web` 服务器。
2. 在服务器的请求处理程序中，我们所做的第一件事是将 `URL` 与我们的路由表进行匹配。 结果将是一个包含服务名称的描述符。
3. 我们从 `consul` 获得实施所需服务的服务器清单。如果这个列表是空的，我们会向客户端返回一个错误。我们使用Tag属性来过滤所有可用的服务，并查找实现当前服务类型的服务器的地址。最后，我们可以将请求路由到它的目的地。 我们根据循环法更新route.index以指向列表中的下一个服务器。然后，我们使用索引从列表中选择一个服务器，并将它与请求（ `req` ）和响应（ `res` ）对象一起传递给 `proxy.web()` 。 这将简单地将请求转发到我们选择的服务器。

现在很清楚，仅使用 `Node.js` 和服务注册表来实现负载均衡器是多么简单，以及我们可以通过这种方式实现多大的灵活性。现在，我们应该准备好了，但首先，请通过以下官方文档安装 `Consul` 服务器： https://www.consul.io/intro/getting-started/install.html 。

这使我们能够通过这个简单的命令行在我们的开发机器中启动 `consul` 服务注册表：

```
consul agent -dev
```

现在我们准备启动负载平衡器：

```
node loadBalancer
```

现在，如果我们尝试访问负载平衡器公开的某些服务，我们会注意到它返回回一个 `HTTP 502` 错误，因为我们还没有启动任何服务器。亲自尝试一下：

```
curl localhost:8080/api
```

上述命令应返回以下输出：

```
Bad Gateway
```

如果我们产生一些服务实例，情况将会发生变化，例如，两个 `api-service` 和一个 `webapp-service` ：

```
forever start app.js api-service
forever start app.js api-service
forever start app.js webapp-service
```

现在负载平衡器应该自动查看新服务器并开始在它们之间分配请求。 让我们尝试使用以下命令：

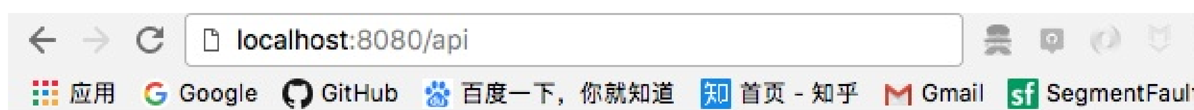
```
curl localhost:8080/api
```

上述命令现在应该返回：

```
api-service response from 6972
```

通过再次运行它，我们现在应该从另一台服务器收到一条消息，确认请求正在不同服务器之间负载均衡：

```
api-service response from 6979
```



api–service response from 6972

这种模式的优点是显而易见的。我们现在可以动态，按需或基于时间表调整我们的基础架构，我们的负载均衡器将自动根据新配置进行调整，无需任何额外的工作！

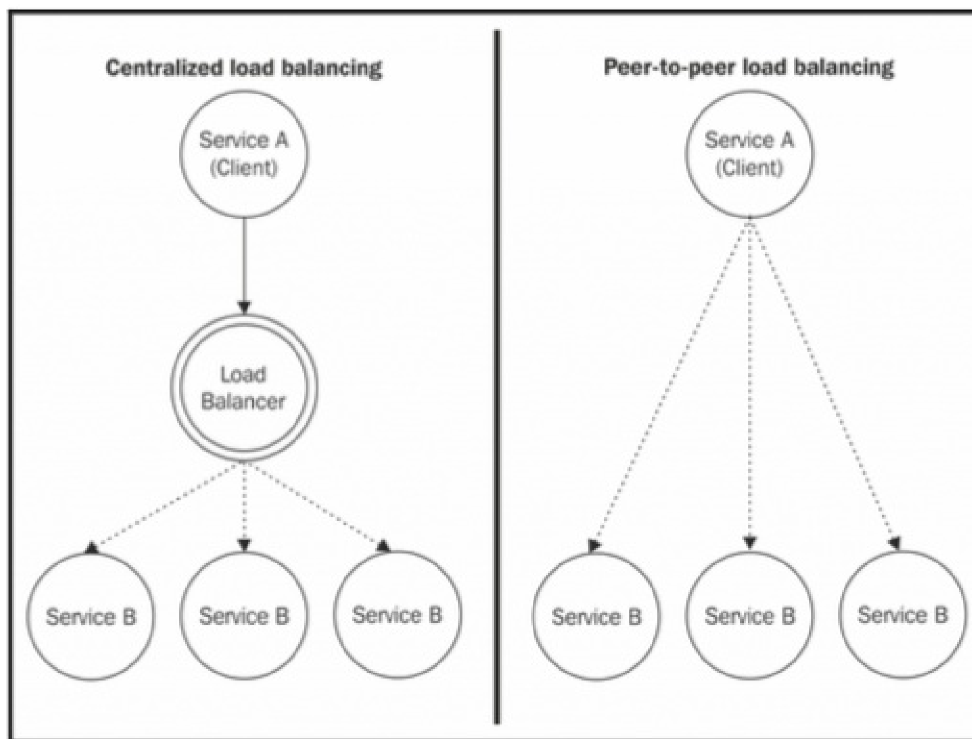
点对点负载平衡

当我们想要将一个复杂的内部网络架构暴露给公共网络（如 `Internet` ）时，使用反向代理几乎是必需的。它有助于隐藏复杂性，提供外部应用程序可轻松使用和依赖的单一访问点。但是，如果我们需要扩展仅供内部使用的服务，则我们可以拥有更多的灵活性和控制力。

假设有一个服务A依靠服务B来实现其功能。服务B在多台机器上进行缩放，并且只能在内部网络中使用。我们迄今为止所了解到的是，服务A将使用反向代理连接到服务B，反向代理会将流量分发到实施服务B的所有服务器。

但是，还有一个选择。我们可以从图中删除反向代理，并直接从客户端（服务A）分发请求，该客户端现在直接负责跨服务B的各种实例负载平衡其连接。只有服务器A知道详细信息关于暴露服务B的服务器，并且在内部网络中，这通常是已知信息。通过这种方法，我们基本上实现了对等负载均衡。

下图比较了我们刚刚描述的两种替代方案：



这是一种非常简单而有效的模式，可以实现真正的分布式通信，而不会出现瓶颈或单点故障。除此之外，它还执行以下操作：

- 通过删除网络节点来降低基础设施的复杂性
- 更快的通信，因为消息将通过更少的节点
- 规模更好，因为性能不受负载平衡器可以处理的限制

另一方面，通过删除反向代理，我们实际上暴露了其底层基础架构的复杂性。此外，通过实施负载平衡算法，每个客户端都必须变得更加智能，并且可能也是保持其基础架构最新的一种方式。

> 点对点负载均衡是ØMQ库中广泛使用的一种模式。

实现可以跨多台服务器平衡请求的HTTP客户端

Messaging and Integration Patterns

如果应用程序涉及到分布式系统。在前一章中，我们学习了如何如何通过使用一些简单的架构模式来集成大量的服务，将其分割到多个机器上。为了使其正常工作，所有机器都必须以某种方式进行交互，因此必须整合它们的交互方式。

有两种主要的技术来集成分布式应用程序：一种是使用共享存储，另一种是使用消息在系统节点上传播数据，这里涉及事件和命令模式。后者在扩展分布式系统时确实有用，这也是后一种方式被广泛运用的原因。

消息被用于软件系统的每一层。我们交换消息以在互联网上进行通信，我们可以使用消息将信息发送到使用管道的其他进程，我们可以使用应用程序中的消息作为直接函数调用（命令模式）的替代方法，甚至也可以使用消息与硬件直接交互。用作在组件和系统之间交换信息的方式的任何离散和结构化数据都可以看作是一条消息。但是，在处理分布式体系结构时，消息传递系统用于描述旨在促进网络信息交换的特定类别的解决方案，模式或者说体系结构。

正如我们将看到的，有几种特征表征这些类型的系统。我们可以选择使用代理模式或点对点结构，我们可以使用请求/回复模式或单向通信，也可以使用队列来更可靠地传递消息；消息整合模式的使用范围非常广泛。本章从 `Node.js` 及其生态系统的角度探讨了这些众所周知的模式中最重要的模式。

总而言之，在本章中，我们将学习以下主题：

- 消息传递系统的基本原理
- 发布/订阅模式
- 管道和任务分配模式
- 请求/回复模式

消息传递系统的基本原理

在谈论消息和消息传递系统时，需要考虑四个基本要素，如下：

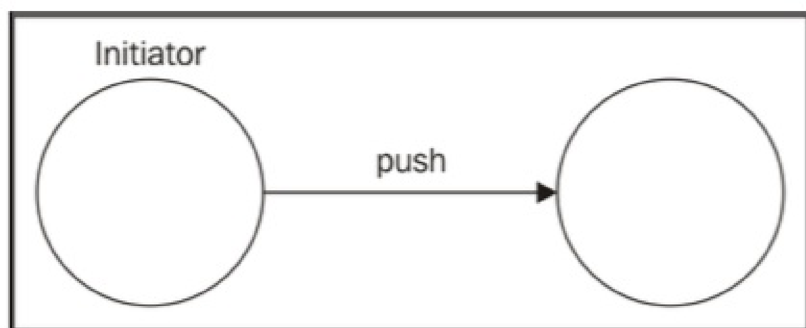
- 通信的方向，可以是单向的，也可以是双向的
- 消息的目的地，这也决定了消息的内容
- 消息的时间，这决定了消息是否可以被立即发送和接收（同步），也可以在将来接收（异步）
- 信息的传递方式，直接传递或通过一个中介者进行传递

在接下来的部分中，我们将把这些方面正式化，以便为我们稍后的讨论奠定基础。

单向通信和请求/回复模式

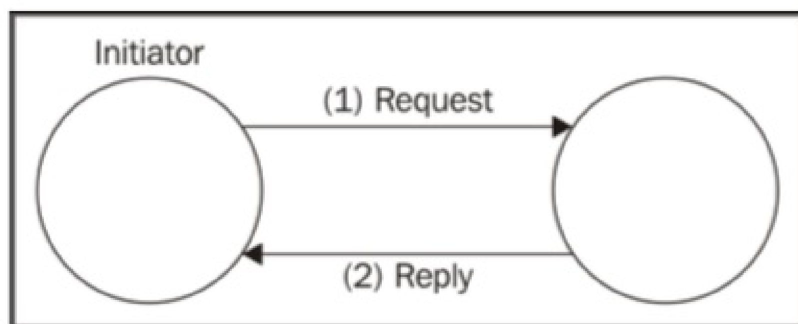
消息传递系统中最基本的方面是通信的传递方向，这个方向通常也表示了这条消息的含义。

最简单的消息传递模式是消息从源到目的地单向推送; 这是一个简单的情况，并不需要太多解释：

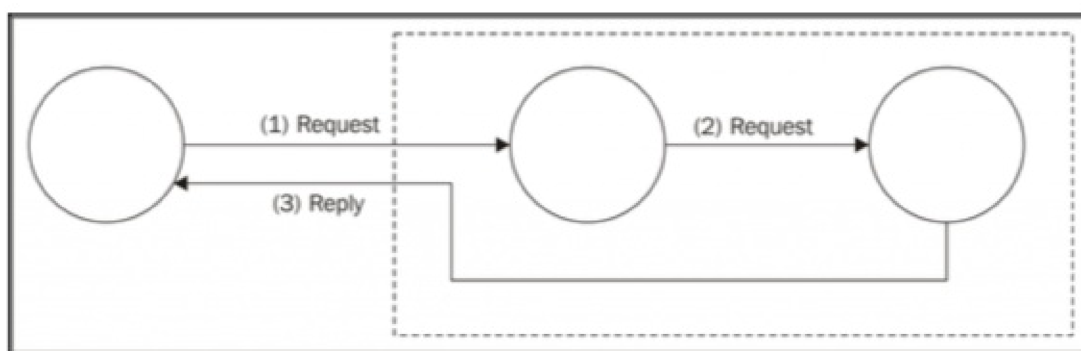


单向通信的一个典型例子是使用 `WebSockets` 向连接的浏览器或 `Web` 服务器发送消息的电子邮件，或将任务分配给一组工作人员的系统。

然而，请求/回复模式比单向通信更受欢迎；一个典型的例子就是调用Web服务。下图显示了这个简单且众所周知的场景：



请求/回复模式可能看起来是一个简单的模式; 但是，当通信异步或涉及多个节点时，我们将看到它变得更加复杂。看看下图中的例子：



通过上图所示的设置，我们可以理解一些请求/回复模式的复杂性。如果我们考虑任何两个节点之间的通信方向，我们可以肯定地说它是单向的。但是，从全局角度来看，发起者发送一个请求，然后接收一个关联的响应，即使来自不同的节点。在这些情况下，真正区分请求/响应模式与单向消息传递模式的区别在于请求和响应之间的关系，它保存在发起者中。回复通常在请求的相同上下文中处理。

消息类型

一条消息本质上是连接不同软件组件的一种方式，这样做的原因有很多：这可能是因为我们想要获得由另一个系统或组件持有的某些信息，或远程执行某项操作，或向某个组件通知某操作刚刚发生。消息内容也会因通信原因而异。 一般来说，我们可以根据消息的目的来确定三种类型的消息：

- 命令消息
- 事件消息
- 文档消息

命令消息

命令消息对我们来说已经很熟悉；它本质上是一个序列化的 `command` 对象，正如我们在 `Chapter 6-Design Patterns` 中所描述的那样。 这种类型的消息的目的是触发 `recevier` 上的动作或任务的执行。为了做到这一点，我们的信息必须包含运行任务的基本信息，这通常是操作的名称和执行时提供的参数列表。 命令消息可用于实现远程过程调用（ `RPC` ）系统，分布式计算或更简单地用于请求某些数据。 `RESTful HTTP` 调用是命令消息的简单示例; 每个 `HTTP` 请求都有一个特定的含义，并与一个精确的操作相关联：例如 `GET` 表示检索资源； `POST` 表示创建一个新的资源； `PUT` 表示更新一个资源； `DELETE` 表示删除一个资源。

事件消息

事件消息用于通知另一个组件发生了某些事件。它通常包含事件的类型，有时还包含一些细节，如 `context` ， `subject` 或 `actor` 。 在 `Web` 开发中，当使用长轮询或 `WebSocket` 接收来自服务器的刚刚发生的事件的通知时，我们在浏览器中使用事件消息，例如数据的变化导致一个时间的发生。事件的使用是分布式应用程序中非常重要的机制，因为它使我们能够将系统的所有节点保持在同一状态上。

文档消息

文档消息主要用于在组件和机器之间传输数据。区分文档消息和命令消息（可能还包含数据）的主要特点是该消息不包含告诉接收方如何处理数据的任何信息。另一方面，与事件消息的主要区别主要是缺少与特定事件的关联。通常，对命令消息的回复是文档消息，因为它们通常只包含请求的数据或操作的结果。

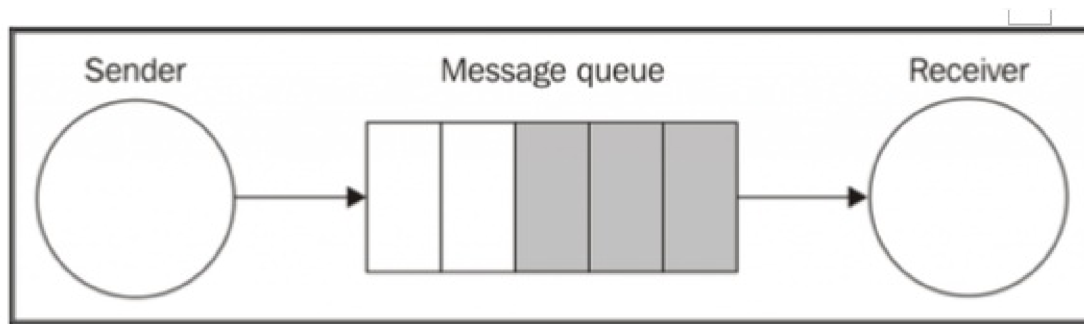
异步消息传递和队列

作为 `Node.js` 开发人员，我们应该已经知道执行异步操作的优势。对于消息和通信而言，这是一回事。

我们可以将同步通信与电话进行比较：两个对等设备必须同时连接到同一个通道，并且它们应该实时交换消息。通常情况下，如果我们想打电话给其他人，我们可能需要另一部手机或关闭正在进行的通信以便开始新的通话。

异步通信类似于 `SMS` ：它不要求收件人在我们发送邮件时连接到网络，我们可能会立即收到回复或者收到未知延迟后的回复，或者我们可能根本没有收到回复。我们可能会将多个 `SMS` 一个接一个地发送给多个收件人，并以任何顺序收到他们的回复（如果有）。简而言之，我们使用更少的资源可以获得更好的并行性。

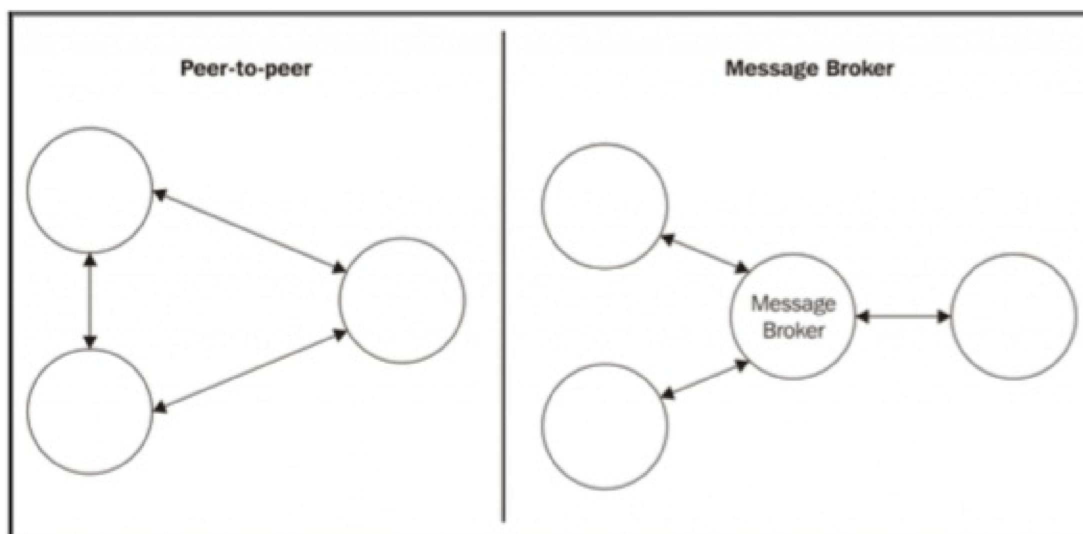
异步通信的另一个重要优点是可以将消息存储并尽快或稍后发送。当接收器太忙而无法处理新消息或我们希望保证传送时，这可能很有用。在消息传递系统中，这可以使用消息队列实现，该消息队列调解发送者和接收者之间的通信，在将消息传递到其目标之前存储任何消息，如下图所示：



如果出于任何原因接收机崩溃，与网络断开连接或速度变慢，则消息会在队列中累积并在接收机联机并且完全正常工作时才可以让发送者继续请求并调度。队列可以位于发送者中，也可以在发送者和接收者之间分开，或者存储在充当通信中间件的专用外部系统中。

点对点或基于代理的消息传递

消息可以以对等方式直接传送给接收方，也可以通过称为消息代理的集中式中介系统传送。代理的主要作用是将发件人的信息接收者分离出来。下图显示了两种方法之间的架构差异：



在对等体系结构中，每个节点都直接负责将消息传递给接收方。这意味着节点必须知道接收方的地址和端口，他们必须就协议和消息格式达成一致。代理从等式中消除了这些复杂性：每个节点都可以完全独立，并且可以与未定义数量的对等进行通信，而无需直接了解其详细信息。 代理还可以充当不同通信协议之间的桥梁，例

如，RabbitMQ broker支持高级消息队列协议（ `AMQP` ），消息队列遥测传输（ `MQTT` ）和 简单/流式文本定向消息协议（ `STOMP` ），支持不同消息协议的多个应用程序进行交互。

> MQTT是一种轻量级消息传递协议，专为机器间通信（物联网）设计。 AMQP是一个更复杂的协议，旨在成为专有消息中间件的开源替代品。STOMP是一个轻量级的基于文本的协议，来自 `HTTP school of design` 。 这三个都是应用层协议，并且基于 `TCP / IP` 。

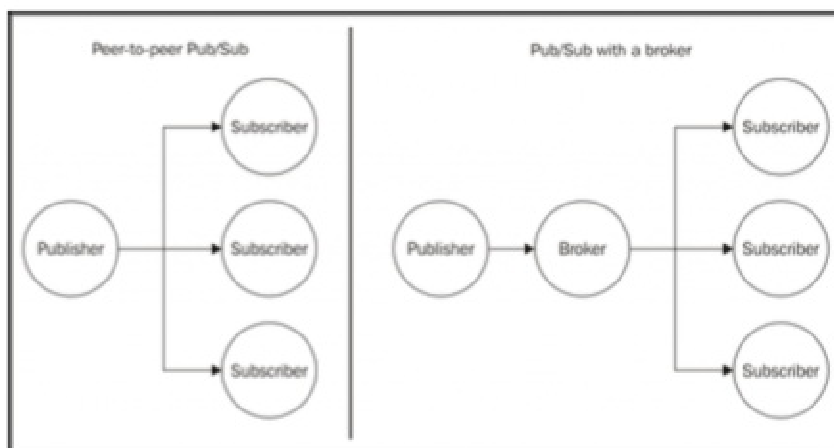
除了解耦和互操作性外，代理还可以提供更多高级功能，如持久队列，路由，消息转换和监控，而不提及许多代理支持的广泛的消息传递模式。当然，没有任何东西可以阻止我们使用对等体系结构实现所有这些功能，但不幸的是，还需要付出更多努力。尽管如此，避免使用代理的原因可能有所不同：

- 代理可能发生故障
- 代理必须扩展，而在对等体系结构中，我们只需要扩展单个节点
- 在没有代理的情况下交换消息可以大大减少传输的延迟

如果我们想要实现一个对等消息传递系统，我们也拥有更多的灵活性和能力，因为我们不受任何特定技术，协议或体系结构的约束。 ØMQ是一个构建消息传递系统的库，其流行性很好地证明了我们可以通过构建定制的对等或混合体系结构获得灵活性。

发布/订阅模式

发布/订阅（通常缩写为 `pub / sub` ）可能是最着名的单向消息传递模式。我们应该已经熟悉它了，因为它不过是一个分布式的观察者模式。就观察者而言，我们有一组用户注册他们对接收特定类别的消息的兴趣。另一方面，发布者产生分布在所有相关用户中的消息。下图显示了发布/订阅模式的两个主要变体，第一个是点对点，第二个使用代理来调解通信：



让pub / sub如此特别的是，发布者不知道邮件的收件人是谁。正如我们所说的那样，用户必须注册它的监听器才能收到特定的消息，从而允许发布者与未知数量的接收者一起工作。换句话说， `pub / sub` 模式的两边是松散耦合的，这使得它成为一个理想模式来集成不断发展的分布式系统的节点。

代理的存在进一步改善了系统节点之间的解耦，因为订阅者仅与代理交互，不知道哪个节点是消息发布者。正如我们稍后将看到的，代理还可以提供消息队列系统，即使在节点之间存在连接问题的情况下也可以实现可靠的传送。

现在，让我们以一个示例来演示这种模式。

构建一个简单的实时聊天应用程序

为了展示 `pub / sub` 模式如何帮助我们集成分布式体系结构的实例，现在我们将使用纯 `WebSockets` 构建一个非常基本的实时聊天应用程序。然后，我们将尝试通过运行多个实例并使用消息传递系统进行通信来扩展它。

实现服务器端

现在，让我们一次一步。 首先构建我们的聊天应用程序; 为此，我们将依赖ws，它是 `Node.js` 的纯 `WebSocket` 实现。我们知道，在 `Node.js` 中实现实时应用程序非常简单，我们的代码将证实这一假设。然后让我们创建聊天的服务器端; 其内容如下（在 `app.js` 文件中）：

```
const WebSocketServer = require('ws').Server;

// 静态的文件服务器
const server = require('http').createServer( //[1]
  require('ecstatic')({
    root: `${__dirname}/www`
  })
);

const wss = new WebSocketServer({
  server: server
}); //[2]
wss.on('connection', ws => {
  console.log('Client connected');
  ws.on('message', msg => { //[3]
    console.log(`Message: ${msg}`);
    broadcast(msg);
  });
});

function broadcast(msg) { //[4]
  wss.clients.forEach(client => {
    client.send(msg);
  });
}

server.listen(process.argv[2] || 8080);
```

这就是我们需要在服务器上实现聊天应用程序的全部内容。这是它的工作方式：

1. 我们首先创建一个 `HTTP` 服务器并附上名为 `ecstatic` 的中间件（ https://npmjs.org/package/ecstatic ）来提供静态文件。 这需要为我们的应用程序（ `JavaScript` 和 `CSS` ）的客户端资源提供服务。
2. 我们创建一个 `WebSocket` 服务器的新实例，并将其附加到我们现有的 `HTTP` 服务器上。然后，我们通过附加连接事件的事件侦听器来开始监听传入的 `WebSocket` 连接。
3. 每当新客户端连接到我们的服务器时，我们就开始监听收到的消息。当新消息到达时，我们将它广播给所有连接的客户端
4. `broadcast()` 函数是对所有连接客户端进行广播， `send()` 函数在其中的每一个客户端上被调用。

这是 `Node.js` 的魔力！ 当然，我们实现的服务器的功能非常少，仅仅实现了基本的功能，但正如我们将看到的，它能够工作。

实现客户端

接下来，是时候实施我们聊天的客户端了;这也是一个非常小而简单的代码片段，基本上是一个包含一些基本 `JavaScript` 代码的最少的 `HTML` 页面。让我们在一个名为 `www/index.html` 的文件中创建这个页面，如下所示：

```
<!DOCTYPE html>
<html>
  <head>
    <script>
      var ws = new WebSocket('ws://' + window.document.location.
host);
      ws.onmessage = function(message) {
        var msgDiv = document.createElement('div');
        msgDiv.innerHTML = message.data;
        document.getElementById('messages').appendChild(msgDiv);
      };

      function sendMessage() {
        var message = document.getElementById('msgBox').value;
        ws.send(message);
      }
    </script>
  </head>
  <body>
    Messages:
    <div id='messages'></div>
    <input type='text' placeholder='Send a message' id='msgBox'>
    <input type='button' onclick='sendMessage()' value='Send'>
  </body>
</html>
```

我们创建的HTML页面并不需要太多解释;它只是一个简单的 `Web` 页面。 我们使用本地 `WebSocket` 对象初始化与 `Node.js` 服务器的连接，然后开始监听来自服务器的消息，并在它们到达时将它们显示在新的 `div` 元素中。相反，我们使用简单的文本框和按钮来发送消息。

在停止或重新启动聊天服务器时， `WebSocket` 连接将关闭，并且不会自动重新连接（如果要实现此则需要使用高级库，例如 `Socket.io` ）。 这意味着在服务器重新启动后重新刷新浏览器以重新建立连接（或实现重新连接机制，这里我们不会介绍）。

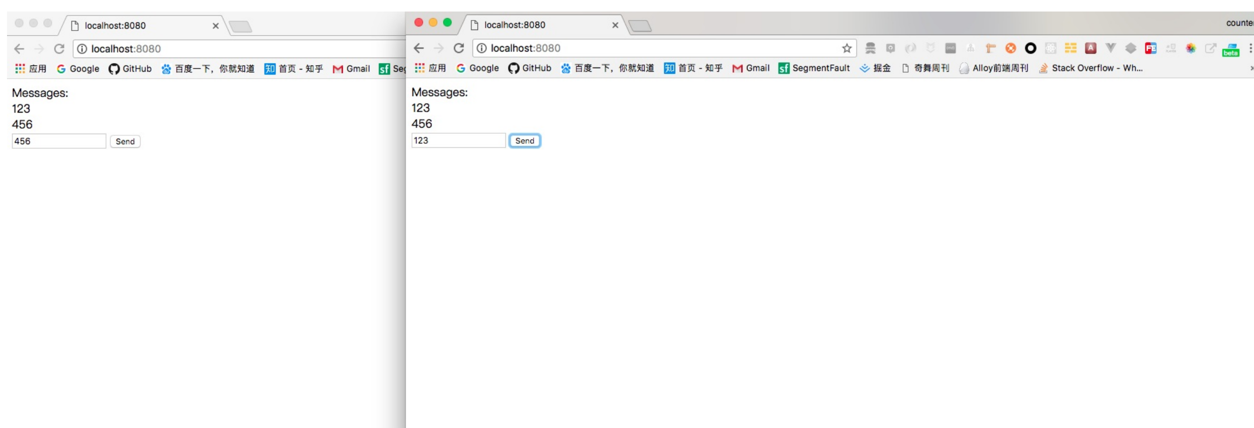
运行和扩展聊天应用程序

我们可以尝试立即运行应用程序;只需使用以下命令启动服务器即可：

```
node app 8080
```

> 要运行这个demo，您需要支持本机 `WebSocket` 的最新浏览器。这里有一个兼容的浏览器列表： http://caniuse.com/#feat=websockets

打开浏览器，访问 http://localhost:8080 ：

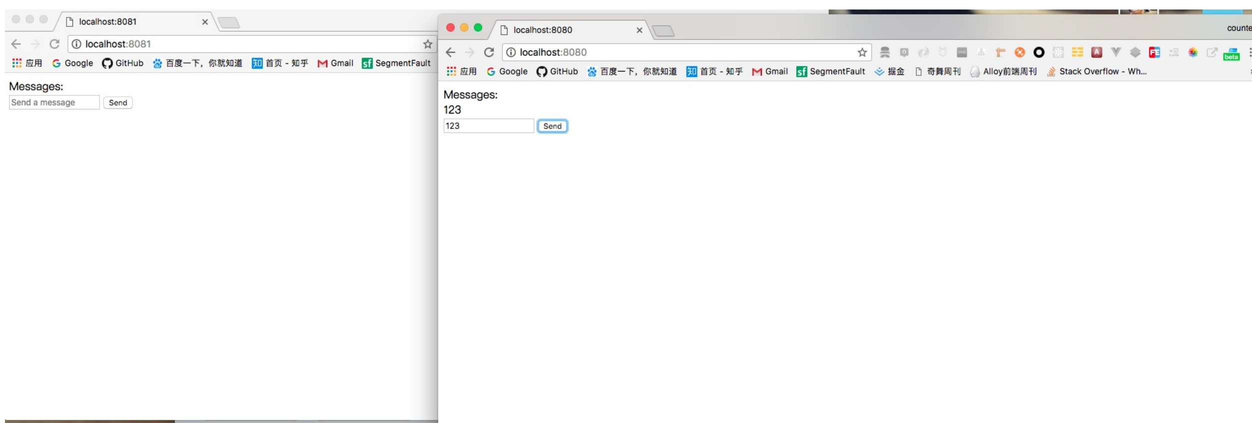


我们现在要展示的是当我们尝试通过启动多个实例来扩展应用程序时发生的情况。 让我们尝试这样做，让我们在另一个端口上启动另一台服务器：

```
node app 8081
```

缩放我们的聊天应用程序的理想结果应该是连接到两个不同服务器的两个客户端应该能够交换聊天消息。不幸的是，这不如我们所愿。 我们可以通过打开另一个浏览器选项卡来尝试打开 http://localhost:8081 。

在一个实例上发送聊天消息时，我们在本地广播一条消息，仅将其分发给连接到该特定服务器的客户端。实际上，两台服务器不会互相通话。 我们需要整合它们。



在实际的应用程序中，我们将使用负载平衡器来分配实例中的负载，但对于此演示，我们不会使用它。这使我们能够以确定性的方式访问每台服务器，以验证它与其它实例交互的方式。

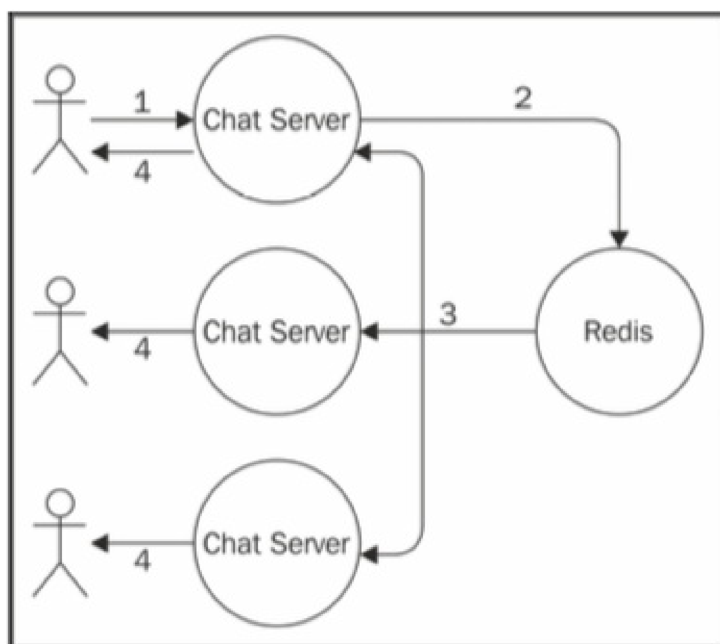
使用Redis作为消息代理

我们通过引入Redis开始分析最重要的 `pub / sub` 实现，这是一个非常快速和灵活的键/值存储，也被许多人定义为数据结构服务器。

Redis比消息代理更像是一个数据库；然而，在其众多功能中，有一对专门用于实现集中式发布/订阅模式的命令。 当然，与更先进的面向消息的中间件相比，这种实现非常简单和基本，但这是其受欢迎的主要原因之一。通常，实际上， `Redis` 已经在现有基础架构中广泛使用，例如，作为缓存服务器或会话存储；它的速度和灵活性使其成为在分布式系统中共享数据的非常流行的选择。因此，只要项目中出现对发布/订阅代理的需求，最简单直接的选择就是重用 `Redis` 本身，避免安装和维护专用的消息代理。让我们以一个例子来展示它的功能。

> 这个例子需要安装 `Redis` ，监听它的默认端口。你可以在这里查看：https://redis.io/topics/quickstart

我们计划使用 `Redis` 来作为聊天服务器的消息代理。每个实例都将从其客户端接收到的任何消息发布给代理，并同时订阅来自其他服务器实例的消息。正如我们所看到的，我们架构中的每个服务器都是订阅者和发布者。下图显示了我们想要获得的体系结构的表示形式：



通过查看上图，我们可以总结一条消息的经历如下：

1. 将消息输入到网页的文本框中并发送到聊天服务器的连接实例。
2. 邮件然后发布给代理。
3. 代理将消息分派给所有订阅者，在我们的体系结构中，所有订阅者都是聊天服务器的实例。
4. 在每种情况下，都会将消息分发给所有连接的客户端。

`Redis` 允许发布和订阅由字符串标识的频道，例如 `chat.nodejs` 。它还允许我们使用 `glob` 风格的模式来定义可能匹配多个频道的订阅，例如 `chat.*` 。

我们在实践中看看它是如何工作的。让我们通过添加发布/订阅逻辑来修改服务器代码：

```
const WebSocketServer = require('ws').Server;
const redis = require("redis");
const redisSub = redis.createClient();
const redisPub = redis.createClient();

// 静态文件服务器
const server = require('http').createServer(
  require('ecstatic')({root: `${__dirname}/www`})
);

const wss = new WebSocketServer({server: server});
wss.on('connection', ws => {
  console.log('Client connected');
  ws.on('message', msg => {
    console.log(`Message: ${msg}`);
    redisPub.publish('chat_messages', msg);
  });
});

redisSub.subscribe('chat_messages');
redisSub.on('message', (channel, msg) => {
  wss.clients.forEach((client) => {
    client.send(msg);
  });
});

server.listen(process.argv[2] || 8080);
```

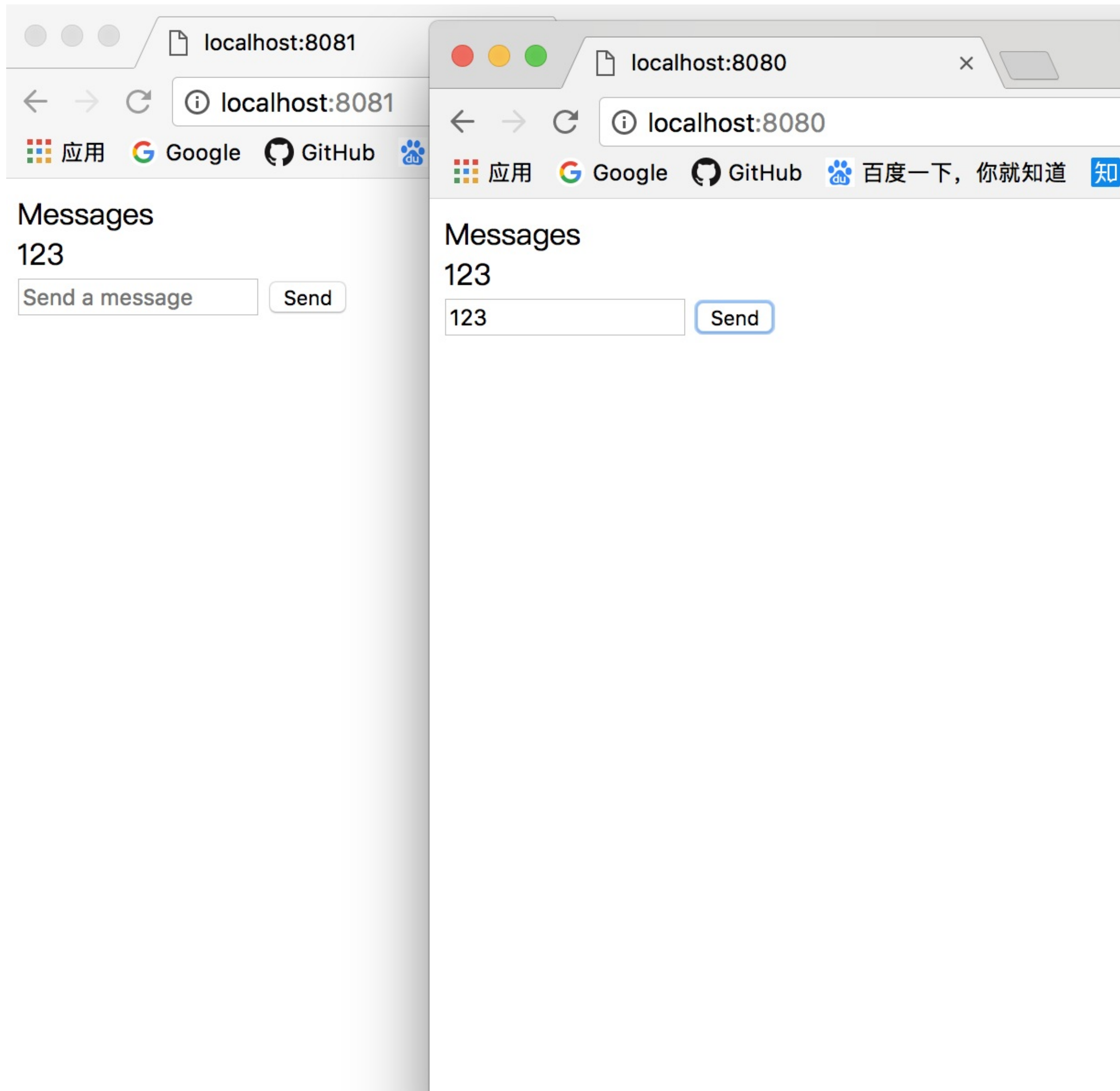
我们对原始聊天服务器所做的更改在前面的代码中突出显示；下面来解释其工作原理：

1. 要将我们的 `Node.js` 应用程序连接到 `Redis` 服务器，我们使用redis，它是一个支持所有可用 `Redis` 命令的完整客户端。 接下来，我们实例化两个不同的连接，一个用于订阅 `channel` ，另一个用于发布消息。 这在 `Redis` 中是必需的，因为一旦连接进入用户模式，就只能使用与订阅相关的命令。 这意味着我们需要第二个连接来发布消息。
2. 当从连接的客户端收到新消息时，我们会在 `chat_messages` 通道中发布消息。我们不直接向客户广播该消息，因为我们所有的服务器订阅了同一个 `channel` （我们稍后会看到），所以它会通过 `Redis` 返回给我们。 对于这个例子的范围来说，这是一个简单而有效的机制。
3. 正如我们所说的，我们的服务器还必须订阅 `chat_messages` 通道，因此我们注册一个侦听器来接收发布到该通道的所有消息（通过当前服务器或任何其他聊天服务器）。当收到消息时，我们只是将它广播给所有连接到当前 `WebSocket` 服务器的客户端。

这些少许的改变足以让聊天服务器信息互通。为了证明这一点，我们可以尝试启动我们应用程序的多个实例：

```
node app 8080
node app 8081
node app 8082
```

然后，我们可以将多个浏览器的选项卡连接到每个实例，并验证我们发送到一台服务器的消息是否被连接到不同服务器的所有其他客户端成功接收。恭喜！我们只使用发布/订阅模式集成了分布式实时应用程序。



```
src/redis-server (redis-server)
o specify a config file use src/redis-server /path/to/redis.conf
26206:M 07 Mar 11:12:27.428 * Increased maximum number of open files to 10032 (it was originally set
to 7168).
                _._
           _.-``__ ''-._
      _.-``    `.  `_.  ''-._           Redis 4.0.8 (00000000/0) 64 bit
  .-`` .-```.  ```\/    _.,_ ''-._
 (    '      ,       .-`  | `,    )     Running in standalone mode
 |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
 |    `-._   `._    /     _.-'    |     PID: 26206
  `-._    `-._  `-./  _.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |           http://redis.io
  `-._    `-._`-.__.-'_.-'    _.-'
 |`-._`-._    `-.__.-'    _.-'_.-'|
 |    `-._`-._        _.-'_.-'    |
  `-._    `-._`-.__.-'_.-'    _.-'
      `-._    `-.__.-'    _.-'
          `-._        _.-'
              `-.__.-'

26206:M 07 Mar 11:12:27.429 # Server initialized
26206:M 07 Mar 11:12:27.429 * Ready to accept connections
```

使用ØMQ进行点对点发布/订阅

代理的存在可以大大简化消息传递系统的体系结构；但是，在某些情况下，它不是最佳解决方案，例如，当不能接受延时的情况下，扩展复杂的分布式系统时，或者当代理节点失败或发生异常的情况。

介绍ØMQ

如果我们的项目可选择点对点消息交换模式，那最佳解决方案应该是ØMQ，也称为 `zmq` 、 `ZeroMQ` 或 `0MQ` ）；我们在本书前面已经提到过这个库。 `ØMQ` 是一个网络库，提供构建各种消息模式的基本工具。它是低级的，速度非常快，并且具有简约的 `API` ，但它提供了消息传递系统的所有基本构建模块，例如原子消息，负载平衡，队列等等。它支持许多类型的传输，例如进程内通道（ `inproc://` ），进程间通信（ `ipc://` ），使用PGM协议（ `pgm://` 或 `epgm://` ）的多播，当然，经典的 `TCP` （ `tcp://` ）。在 `ØMQ` 的功能中，我们还可以找到实现发布/订阅模式的工具，这正是我们的例子所需要的。因此，我们现在要做的是从聊天应用程序的体系结构中删除代理（ `Redis` ），并让各个节点以对等方式进行通信，利用 `ØMQ` 的发布/订阅套接字。

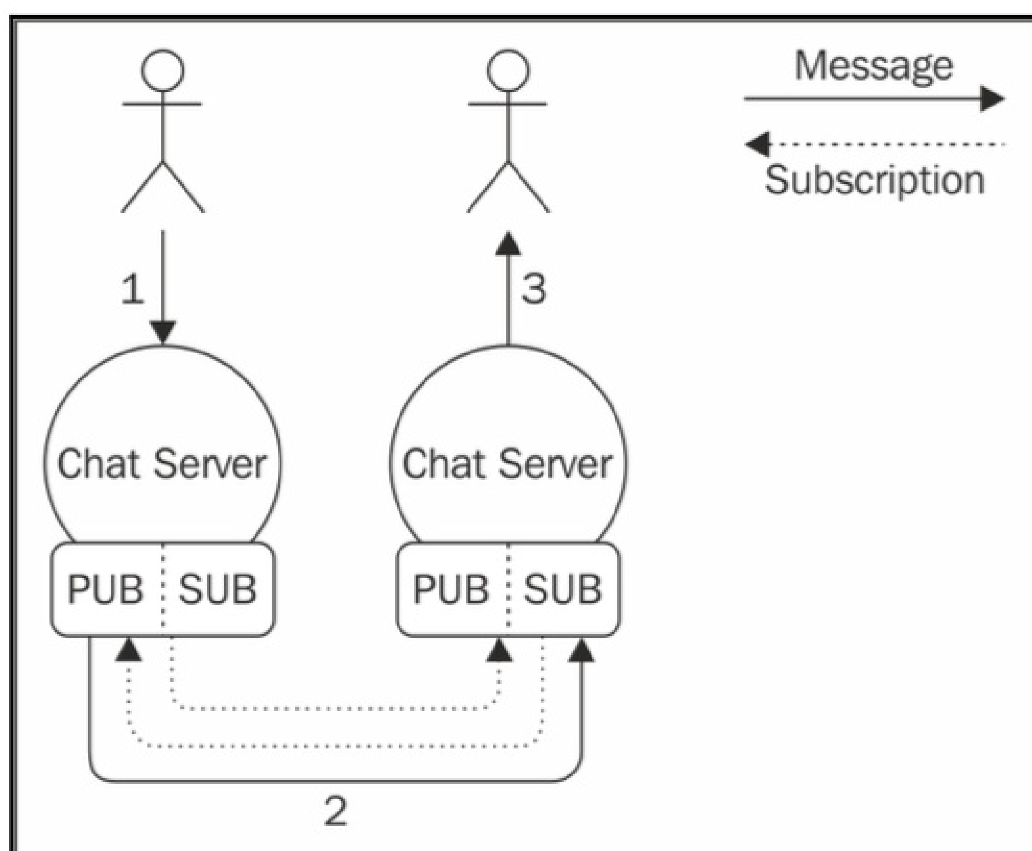
> `ØMQ` 套接字可以被视为类固化网络套接字，它提供了很多方法来帮助实现最常见的消息传递模式。例如，我们可以找到实现发布/订阅，请求/回复或单向通信的套接字。

为聊天设计一个对等体系结构的服务器

当我们从架构中移除代理时，聊天应用程序的每个实例都必须直接连接到其他可用实例，以便接收他们发布的消息。 在ØMQ中，我们有两种专门为此设计的套接字： `PUB` 和 `SUB` 。典型的模式是将 `PUB` 套接字绑定到一个端口，该端口将开始侦听来自其他 `SUB` 套接字的订阅。

订阅可以有一个过滤器，指定将传递到 `SUB` 套接字的消息。该过滤器是一个简单的二进制缓冲区（所以它也可以是一个字符串），它将与消息的开头（这也是一个二进制缓冲区）相匹配。当通过 `PUB` 套接字发送一条消息时，它将被广播到所有连接的 `SUB` 套接字，但仅在应用了它们的订阅过滤器之后。仅当使用连接的协议时，过滤器才会应用到发布方，例如 `TCP` 。

下图显示了应用于我们的分布式聊天服务器体系结构的模式（为简单起见，仅有两个实例）：



> 要运行本节中的示例，您需要在系统上安装本地 `ØMQ` 二进制文件。 你可以在 http://zeromq.org/intro:get-the-software 找到更多信息。注意：此示例已针对 `ØMQ` 的 `4.0` 分支进行了测试。

使用 `ØMQ` 的 `PUB / SUB` 套接字

让我们通过修改我们的聊天服务器来看看它是如何工作的：

```
const WebSocketServer = require('ws').Server;
const args = require('minimist')(process.argv.slice(2));
const zmq = require('zmq');

//static file server
const server = require('http').createServer(
  require('ecstatic')({root: `${__dirname}/www`})
);

const pubSocket = zmq.socket('pub');
pubSocket.bind(`tcp://127.0.0.1:${args['pub']}`);

const subSocket = zmq.socket('sub');
const subPorts = [].concat(args['sub']);
subPorts.forEach(p => {
  console.log(`Subscribing to ${p}`);
  subSocket.connect(`tcp://127.0.0.1:${p}`);
});
subSocket.subscribe('chat');

subSocket.on('message', msg => {
  console.log(`From other server: ${msg}`);
  broadcast(msg.toString().split(' ')[1]);
});

const wss = new WebSocketServer({server: server});
wss.on('connection', ws => {
  console.log('Client connected');
  ws.on('message', msg => {
    console.log(`Message: ${msg}`);
    broadcast(msg);
    pubSocket.send(`chat ${msg}`);
  });
});

function broadcast(msg) {
  wss.clients.forEach(client => {
    client.send(msg);
  });
}

server.listen(args['http'] || 8080);
```

前面的代码清楚地表明，我们的应用程序的逻辑变得稍微复杂一些；然而，考虑到我们正在实施分布式和点对点的发布/订阅模式，它仍然很简单。让我们看看所有的部分是如何结合在一起的：

1. 我们需要zmq，它基本上是 `ØMQ` 库的 `Node.js` 版本。我们还需要minimist，它是一个命令行参数解析器；我们需要这个能够轻松接受命名参数。
2. 我们立即创建我们的 `PUB` 套接字并将其绑定到 - `pub` 命令行参数中提供的端口。

3. 我们创建 `SUB` 套接字，并将它连接到应用程序其他实例的 `PUB` 套接字。目标 `PUB` 套接字的端口在-- `sub` 命令行参数中提供（可能有多个）。然后，我们通过提供 `chat` 作为过滤器来创建实际订阅，这意味着我们只会收到以 `chat` 开始的消息。
4. 当我们的 `WebSocket` 接收到新消息时，我们将它广播给所有连接的客户端，但我们也通过 `PUB` 套接字发布它。 我们使用 `chat` 作为前缀，然后是空格，因此该消息将作为过滤器发布到所有使用 `chat` 的订阅者。
5. 我们开始监听到达我们 `SUB` 套接字的消息，我们对消息做一些简单的解析以删除聊天前缀，然后我们将它广播给所有连接到当前 `WebSocket` 服务器的客户端。

我们现在已经构建了一个简单的分布式系统，使用点对点发布/订阅模式进行集成！

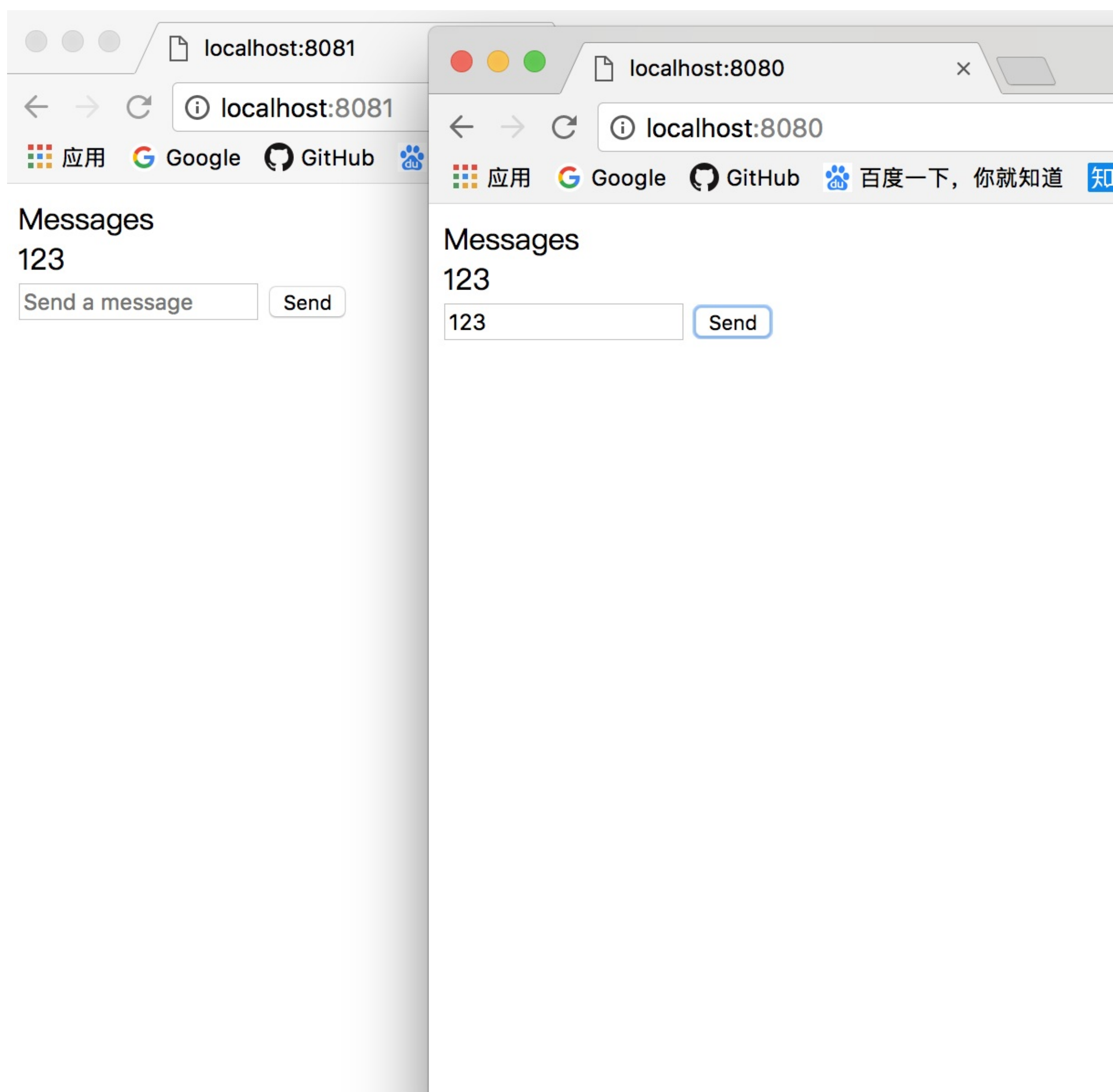
让我们开始吧，让我们通过确保正确连接它们的 `PUB` 和 `SUB` 插槽来启动我们的应用程序的三个实例：

```
node app --http 8080 --pub 5000 --sub 5001 --sub 5002
node app --http 8081 --pub 5001 --sub 5000 --sub 5002
node app --http 8082 --pub 5002 --sub 5000 --sub 5001
```

第一个命令将启动一个 `HTTP` 服务器侦听端口 `8080` 的实例，同时在端口 `5000` 上绑定 `PUB` 套接字，并将 `SUB` 套接字连接到端口 `5001` 和 `5002` ，这是其他两个实例的 `PUB` 套接字应该侦听的端口。其他两个命令以类似的方式工作。

现在，我们可以看到的第一件事情是，如果与 `PUB` 套接字对应的端口不可用， `ØMQ` 不会崩溃。例如，在第一个命令执行时，端口 `5001` 和 `5002` 仍然不可用；但是， `ØMQ` 不会引发任何错误。这是因为 `ØMQ` 具有重连机制，它会自动尝试定期与这些端口建立连接。如果任何节点出现故障或重新启动，此功能特别适用。相同的逻辑适用于 `PUB` 套接字：如果没有订阅者，它将简单地删除所有消息，但它将继续工作。

此时，我们可以尝试使用浏览器导航到我们启动的任何服务器实例，并验证这些消息是否适当地向所有聊天服务器广播。



在前面的例子中，我们假设了一个静态体系结构，其中实例的数量和地址是事先已知的。我们可以引入一个服务注册表，如前一章所述，动态连接我们的实例。同样重要的是要指出 `ØMQ` 可以用来实现代理模式。

持久订阅者

消息传递系统中的一个重要抽象是消息队列（ `MQ` ）。对于消息队列，消息的发送者和接收者不需要同时处于活动状态和连接状态以建立通信，因为排队系统负责存储消息直到目的地能够 接收他们。 这种行为与 `set and forget` 范式相反，订户只能在消息系统连接期间才能接收消息。

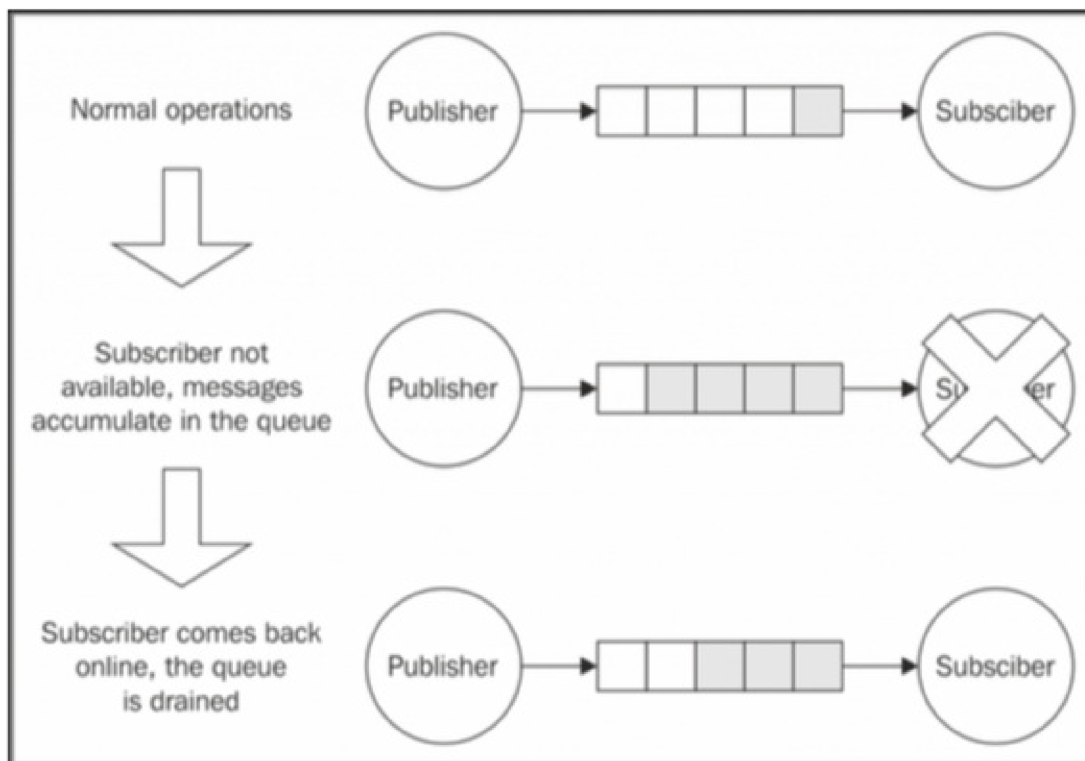
一个能够始终可靠地接收所有消息的订阅者，即使是在没有收听这些消息时发送的消息，也被称为持久订阅者。

> `MQTT` 协议为发送方和接收方之间交换的消息定义了服务质量（QoS）级别。这些级别对描述任何其他消息系统（不仅仅是 `MQTT` ）的可靠性也非常有用。如下描述：

`QoS0` ，最多一次：也被称为“设置并忘记”，消息不会被保留，并且传送未被确认。这意味着在接收机崩溃或断开的情况下，信息可能会丢失。 `QoS1` ，至少一次：保证至少收到一次该消息，但如果在通知发件人之前接收器崩溃，则可能发生重复。这意味着消息必须在必须再次发送的情况下持续下去。 `QoS2` ，正好一次：这是最可靠的 `QoS` ；它保证该消息只被接收一次。 这是以用于确认消息传递的更慢和更数据密集型机制为代价的。

> 请在MQTT规范中了解更多信息 http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt /mqtt-v3r1.html#qos-flows

正如我们所说的，对于持久订阅者，我们的系统必须使用消息队列来在用户断开连接时累积消息。队列可以存储在内存中，也可以保存在磁盘上以允许恢复其消息，即使代理重新启动或崩溃。下图显示了由消息队列支持的持久订阅者：



持久订阅者可能是消息队列所支持的最重要的模式，但它肯定不是唯一的模式，我们将在本章后面看到。

`Redis` 的发布/订阅命令实现了一个设置和遗忘机制（ `QoS0` ）。但是， `Redis` 仍然可以用于使用其他命令的组合来实现持久订阅者（不直接依赖其发布/订阅实现）。您可以在以下博客文章中找到关于此技术的说明：

- https://davidmarquis.wordpress.com/2013/01/03/reliable-delivery-message-queues-with-redis/
- http://www.ericjperry.com/redis-message-queue/

`ØMQ` 定义了一些支持持久订阅者的模式，但实现这种机制主要取决于我们。

介绍 `AMQP`

消息队列通常用于不能丢失消息的情况，其中包括任务关键型应用程序，如银行或金融系统。这通常意味着典型的企业级消息队列是一个非常复杂的软件，它使用 `bulletproof protocols` 和持久存储来保证即使在出现故障时也能传送消息。由于这个原因，企业消息传递中间件多年来一直是 `Oracle` 和 `IBM` 等巨头的特权，它们中的每一个通常都实施自己的专有协议，导致强大的客户锁定。幸运的是，由于诸如 `AMQP` ， `STOMP` 和 `MQTT` 等开放协议的增长，邮件系统进入主流已经有几年了。为了理解消息队列系统的工作原理，现在我们将概述 `AMQP` ；这是了解如何使用基于此协议的典型 `API` 的基础。

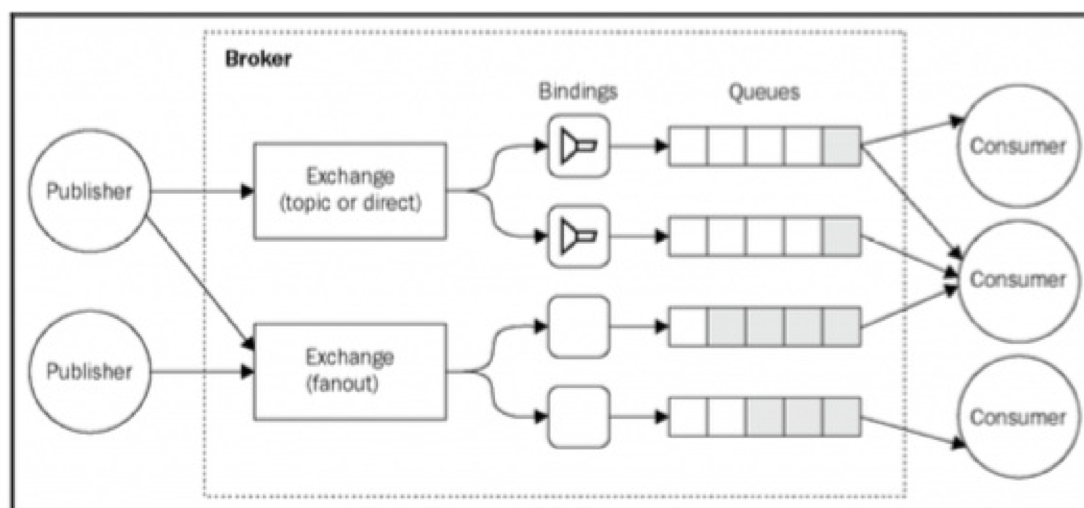
`AMQP` 是许多消息队列系统支持的开放标准协议。除了定义通用通信协议外，它还提供了描述路由，过滤，排队，可靠性和安全性的模型。在 `AMQP` 中，有三个基本组成部分：

- `Queue（队列）` ：负责存储客户端消费的消息的数据结构。我们的应用程序推送消息到队列，供给一个或多个消费者。如果多个使用者连接到同一个队列，则这些消息会在它们之间进行负载平衡。 队列可以是以下之一：
  - `Durable（持久队列）` ：这意味着如果代理重新启动，队列会自动重新创建。一个持久的队列并不意味着它的内容也被保留下来；实际上，只有标记为持久性的消息才会保存到磁盘，并在重新启动的情况下进行恢复。
  - `Exclusive（专有队列）` ：这意味着队列只能绑定到一个特定的用户连接。当连接关闭时，队列被销毁。
  - `Auto-delete（自动删除队列）` ：这会导致队列在最后一个用户断开连接时被删除。
- `Exchange（交换机）` ：这是发布消息的地方。交换机根据它实现的算法将消息路由到一个或多个队列：
  - `Direct exchange（直接交换机）` ：通过匹配路由键（例如， `chat.msg` ）整个消息来路由消息。
  - `Topic exchange（主题交换机）` ：它使用与路由密钥相匹配的类似 `glob` 的模式分发消息（例如， `chat.#` 匹配以 `chat` 开始的所有路由密钥）。
  - `Fanout exchange（扇出交换机）` ：它向所有连接的队列广播消息，忽略提供的任何路由密钥。
- `Binding（绑定）` ：这是交换机和队列之间的链接。它还定义了路由键或用于过滤从交换机到达的消息的模式。

这些组件由代理管理，该代理公开用于创建和操作它们的 `API` 。当连接到代理时，客户端创建一个到连接的通道，负责维护与代理的通信状态。

> 在 `AMQP` 中，可以通过创建任何类型的非排他性或自动删除的队列来获得持久用户模式。

下图将所有这些组件放在一起：



`AMQP` 模型比我们目前使用的消息系统（ `Redis` 和 `ØMQ` ）更复杂；但是，比起只使用原生发布/订阅机制，它提供了一系列功能和可靠性的保证。

> 您可以在 `RabbitMQ` 网站上找到 `AMQP` 模型的详细介绍：
> https://www.rabbitmq.com/tutorials/amqp-concepts.html

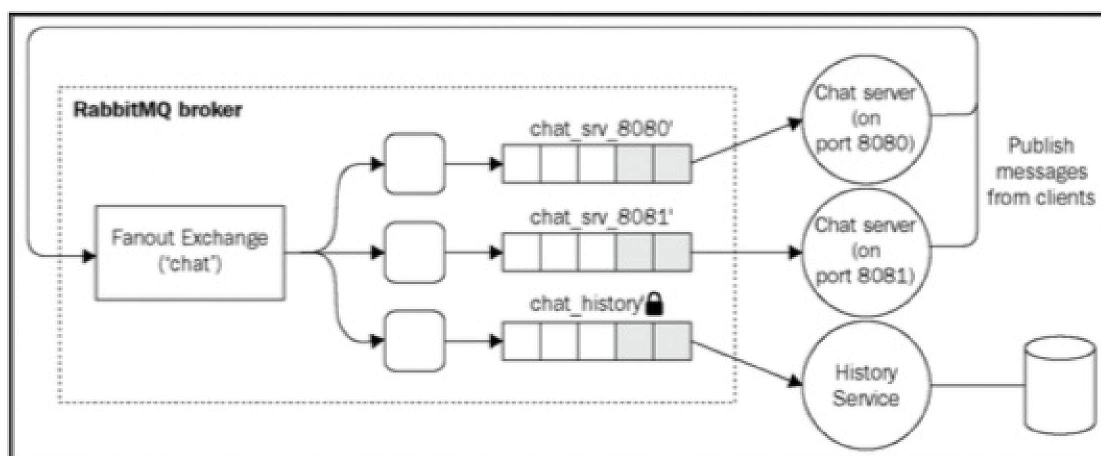
使用 `AMQP` 和 `RabbitMQ` 的持久订阅者

现在让我们练习一下我们了解持久订阅者和 `AMQP` 的内容，并开发一个小例子。不丢失任何消息很重要的典型场景是，我们希望保持微服务体系结构的不同服务同步；我们在前一章已经描述了这种集成模式。如果我们想要使用经纪商将所有服务保留在同一页面上，那么我们不会丢失任何信息是非常重要的，否则我们可能会处于不一致的状态。

为聊天应用程序设计一个历史记录服务

现在让我们使用微服务方法扩展我们的小聊天应用程序。让我们添加一个历史记录服务，将我们的聊天消息保存在数据库中，这样当客户端连接时，我们可以查询服务并检索整个聊天记录。我们将使用RabbitMQ broker和 `AMQP` 将历史记录服务器与聊天服务器相集成。

下图显示了我们的架构：



如前面的体系结构所述，我们将使用单个扇出交换机；我们不需要任何特定的路由，所以我们的场景不需要任何更复杂的交换。接下来，我们将为聊天服务器的每个实例创建一个队列。这些队列是排他性的；当聊天服务器处于脱机状态时，我们无意收到任何遗漏的消息，都会传送给历史记录服务器记录，最终还可以针对存储的消息实施更复杂的查询。实际上，这意味着我们的聊天服务器不是持久订阅者，并且只要连接关闭，它们的队列就会被销毁。

相反，历史记录服务器不能丢失任何信息; 否则，它不会达到其目的。我们要为它创建的队列必须耐用，以便在历史记录服务断开连接时发布的任何消息将保留在队列中，并在联机时交付。

我们将使用熟悉的 `LevelUP` 作为历史记录服务的存储引擎，而我们将使用amqplib，并通过 `AMQP` 协议连接到 `RabbitMQ` 。

> 以下示例需要工作的 `RabbitMQ` 服务器，侦听其默认端口。 有关更多信息，请参阅其官方安装指南： http://www.rabbitmq.com/download.html

使用 **`AMQP`** 实现可靠的历史记录服务

现在让我们实施我们的历史记录服务器！我们将创建一个独立的应用程序（典型的微服务），它在模块 `historySvc.js` 中实现。该模块由两部分组成：向客户端展示聊天记录的 `HTTP` 服务器，以及负责捕获聊天消息并将其存储在本地数据库中的 `AMQP` 使用者。

让我们来看看下面代码中的内容：

```
const level = require('level');
const timestamp = require('monotonic-timestamp');
const JSONStream = require('JSONStream');
const amqp = require('amqplib');
const db = level('./msgHistory');

require('http').createServer((req, res) => {
  res.writeHead(200);
  db.createValueStream()
    .pipe(JSONStream.stringify())
    .pipe(res);
}).listen(8090);

let channel, queue;
amqp
  .connect('amqp://localhost')  // [1]
  .then(conn => conn.createChannel())
  .then(ch => {
    channel = ch;
    return channel.assertExchange('chat', 'fanout');  // [2]
  })
  .then(() => channel.assertQueue('chat_history'))  // [3]
  .then((q) => {
    queue = q.queue;
    return channel.bindQueue(queue, 'chat');  // [4]
  })
  .then(() => {
    return channel.consume(queue, msg => {  // [5]
      const content = msg.content.toString();
      console.log(`Saving message: ${content}`);
      db.put(timestamp(), content, err => {
        if (!err) channel.ack(msg);
      });
    });
  })
  .catch(err => console.log(err))
;
```

我们可以立即看到 `AMQP` 需要一些设置，这对创建和连接模型的所有组件都是必需的。观察 `amqplib` 默认支持 `Promises` 也很有趣，所以我们大量利用它们来简化应用程序的异步步骤。让我们详细看看它是如何工作的：

1. 我们首先与 `AMQP` 代理建立连接，在我们的例子中是 `RabbitMQ` 。然后，我们创建一个 `channel` ，该 `channel` 类似于保持我们通信状态的会话。
2. 接下来，我们建立了我们的会话，名为 `chat` 。正如我们已经提到的那样，这是一种扇出交换机。 `assertExchange()` 命令将确保代理中存在交换，否则它将创建它。
3. 我们还创建了我们的队列，名为 `chat_history` 。默认情况下，队列是持久的；不是排他性的，也不会自动删除，所以我们不需要传递任何额外的选项来支持持久订阅者。

4. 接下来，我们将队列绑定到我们以前创建的交换机。在这里，我们不需要任何其他特殊选项，例如路由键或模式，因为交换机是扇出类型的交换机，所以它不执行任何过滤。
5. 最后，我们可以开始监听来自我们刚创建的队列的消息。我们将使用时间戳记作为密钥（ https://npmjs.org/package/monotonic-timestamp ）在 `LevelDB` 数据库中收到的每条消息保存，以保持消息按日期排序。看到我们使用 `channel.ack(msg)` 来确认每条消息，并且只有在消息成功保存到数据库后，也很有趣。如果代理没有收到 `ACK` （确认），则该消息将保留在队列中以供再次处理。这是 `AMQP` 将服务可靠性提升到全新水平的另一个重要特征。如果我们不想发送明确的确认，我们可以将选项 `{noAck:true}` 传递给 `channel.consume() API` 。

将聊天应用程序与 **AMQP** 集成

要使用 `AMQP` 集成聊天服务器，我们必须使用与我们在历史记录服务器中实现的设置非常相似的设置，因此我们不打算在此重复。 但是，看看队列是如何创建的以及如何将新消息发布到交换中仍然很有趣。新的 `app.js` 文件的相关部分如下：

```
const WebSocketServer = require('ws').Server;
const amqp = require('amqplib');
const JSONStream = require('JSONStream');
const request = require('request');
let httpPort = process.argv[2] || 8080;

const server = require('http').createServer(
  require('ecstatic')({root: `${__dirname}/www`})
);

let channel, queue;
amqp
  .connect('amqp://localhost')
  .then(conn => conn.createChannel())
  .then(ch => {
    channel = ch;
    return channel.assertExchange('chat', 'fanout');
  })
  .then(() => {
    return channel.assertQueue(`chat_srv_${httpPort}`, {exclusiv
e: true});
  })
  .then(q => {
    queue = q.queue;
    return channel.bindQueue(queue, 'chat');
  })
  .then(() => {
    return channel.consume(queue, msg => {
      msg = msg.content.toString();
      console.log('From queue: ' + msg);
      broadcast(msg);
    }, {noAck: true});
```

```
  })
  .catch(err => console.log(err))
;

const wss = new WebSocketServer({server: server});
wss.on('connection', ws => {
  console.log('Client connected');
  //query the history service
  request('http://localhost:8090')
    .on('error', err => console.log(err))
    .pipe(JSONStream.parse('*'))
    .on('data', msg => ws.send(msg))
  ;

  ws.on('message', msg => {
    console.log(`Message: ${msg}`);
    channel.publish('chat', '', new Buffer(msg));
  });
});

function broadcast(msg) {
  wss.clients.forEach(client => client.send(msg));
}

server.listen(httpPort);
```

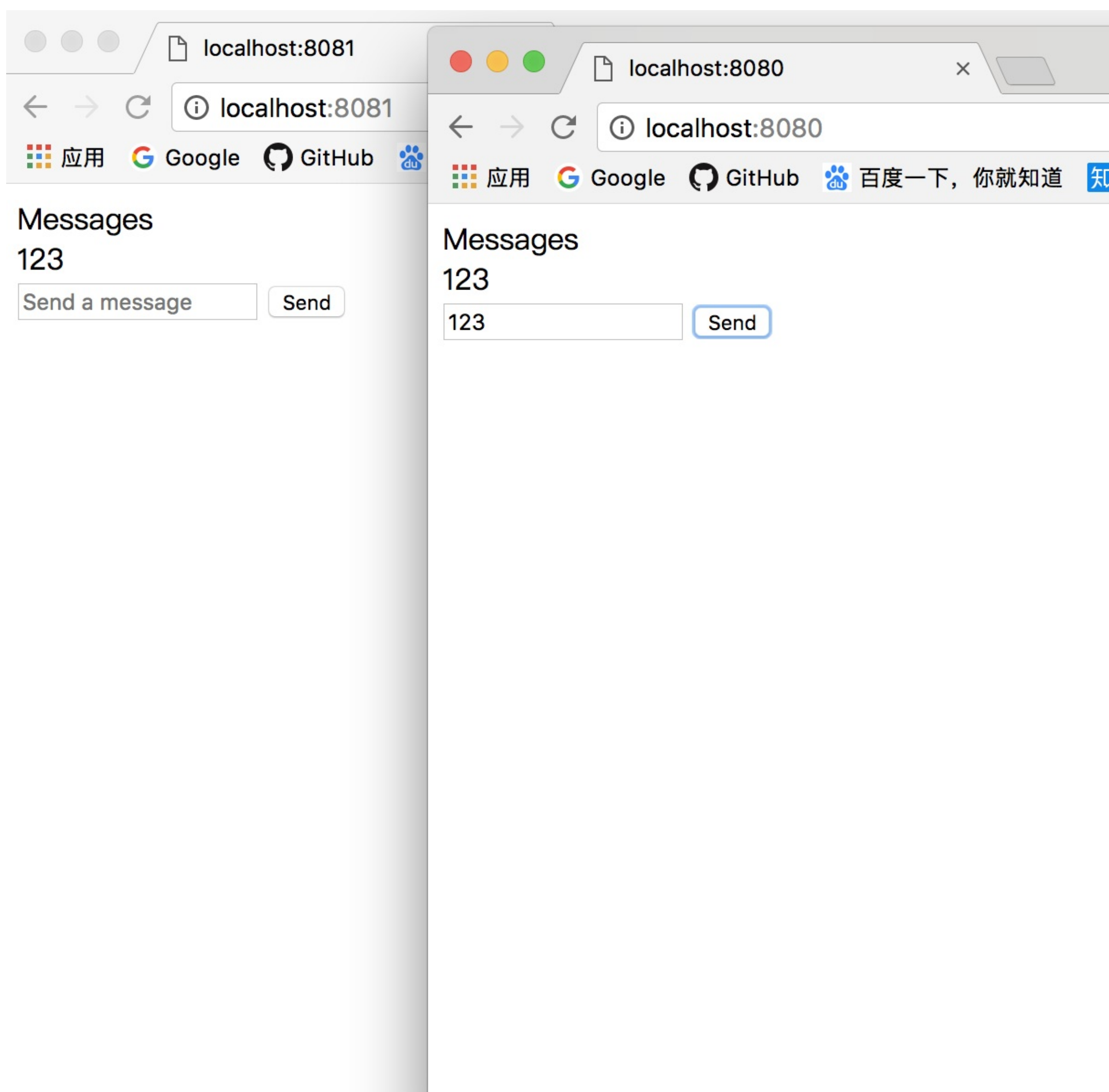
正如我们所提到的，我们的聊天服务器不需要成为持久的订阅者。 所以当我们创建我们的队列时，我们传递选项 `{exclusive:true}` ，指示队列被限制到当前连接，因此一旦聊天服务器关闭，它就会被销毁。

发布新消息也很容易; 我们只需要指定目标交换机（聊天）和一个路由键，在我们的情况下这是空的（"），因为我们正在使用扇出交换。

我们现在可以运行我们改进的聊天应用程序架构；为此，我们开始两个聊天服务器和历史服务：

```
node app 8080
node app 8081
node historySvc
```

现在看看我们的系统，特别是历史服务如何在停机的情况下运行，这一点很有意思。如果我们停止历史记录服务器并继续使用聊天应用程序的 `Web UI` 发送消息，我们将会看到，当历史记录服务器重新启动时，它将立即收到所有错过的消息。

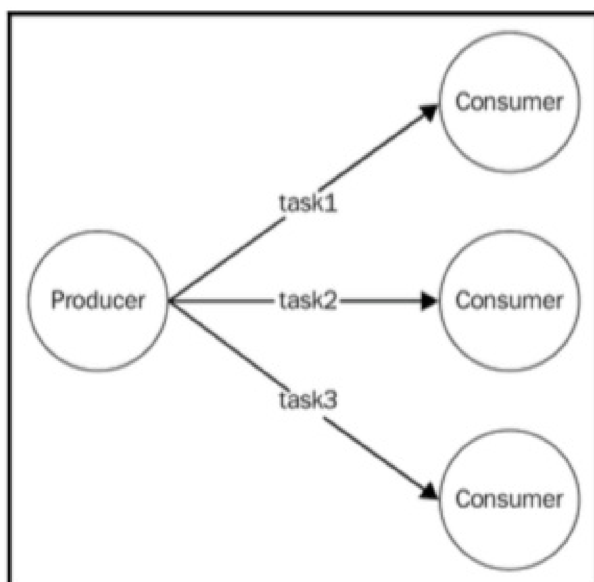


管道和任务分配模式

在 `Chapter9-Advanced Asynchronous Recipes` 中，我们学习了如何将高耗能的任务委派给多个本地进程，但即使这是一种有效的方法，但它也无法在单个机器的边界之外进行缩放。在本节中，我们将看到如何在分布式架构中使用类似的模式，使用位于网络中任何位置的远程 `worker` 。

这个想法是有一个消息传递模式，允许我们跨多台机器传播任务。 这些任务可能是单独的工作块或者使用分而治之技术分割的更大任务。

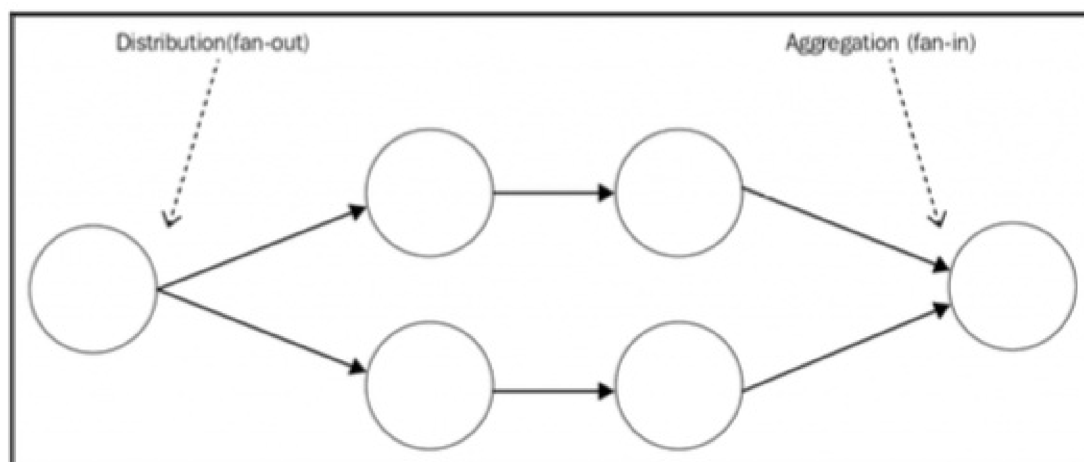
如果我们看看下图所示的逻辑架构，我们应该能够认识到一种熟悉的模式：



从上图我们可以看到，发布/订阅模式不适合这种类型的应用程序，因为我们绝对不希望多个 `worker` 接收任务。 我们需要的是一种类似于负载均衡器的消息分发模式，它将每条消息分派给不同的消费者（在本例中也称为worker）。在消息传递系统术语中，这种模式被称为竞争消费者。

与上一章中我们看到的 `HTTP` 负载均衡器的一个重要区别是，在这里，消费者扮演着更积极的角色。事实上，我们将在后面看到，大多数情况下，生产者不是连接到消费者，而是连接到任务生产者或任务队列的消费者本身，以便接收新的工作。这对于可扩展系统来说是一个很大的优势，因为它可以在不修改生产者或采用服务注册表的情况下无缝增加 `worker` 数量。

另外，在通用消息传递系统中，我们不一定需要生产者和 `worker` 之间的请求/回复通信。相反，大多数情况下，首选的方法是使用单向异步通信，这可以实现更好的并行性和可伸缩性。在这样的体系结构中，消息可能总是以一个方向行进，创建管道，如下图所示：



管道允许我们构建非常复杂的处理体系结构，而不需要同步请求/应答通信的负担，通常导致更低的延迟和更高的吞吐量。 在上图中，我们可以看到消息如何在一组 `worker` 分布，并被转发到其他处理单元，然后聚合到通常称为接收器的单个节

点（扇入）中。

在本节中，我们将通过分析两个最重要的变体，即点对点通信和代理模式为基础，来关注这些架构的构建。

> 管道与任务分配模式的组合也称为并行管道。

ØMQ扇出/扇出模式

我们已经发现了 `ØMQ` 在构建点对点分布式体系结构方面的一些优势。在前一节中，我们使用 `PUB` 和 `SUB` 套接字向多个消费者传播单个消息；现在我们将看到如何使用称为 `PUSH` 和 `PULL` 的另一对套接字来构建并行管道。

PUSH/PULL套接字

直观地说，我们可以说 `PUSH` 套接字用于发送消息，而 `PULL` 套接字是用于接收的。这似乎是一个微不足道的组合；然而，它们有一些很好的特性，使它们成为构建单向通信系统的完美选择：

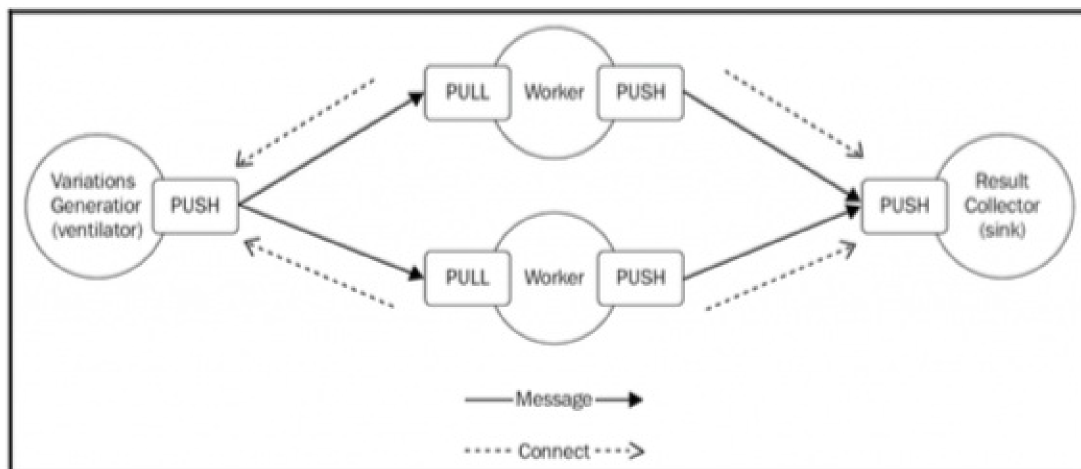
- 两者都可以在 `connet` 模式或 `bind` 模式下工作。换句话说，我们可以构建一个 `PUSH` 套接字并将其绑定到本地端口，以监听来自 `PULL` 套接字的传入连接，反之亦然， `PULL` 套接字可以监听来自 `PUSH` 套接字的连接。消息总是以相同的方向传播，从 `PUSH` 到 `PULL` ；它只是连接的发起者可能是不同的。绑定模式是耐用节点（例如任务生产者和接收器）的最佳解决方案，而连接模式对于瞬态节点（例如任务工作者）来说是完美的。这使得瞬时节点的数量可以任意变化，而不会影响其它正在使用的节点。
- 如果有多个 `PULL` 套接字连接到单个 `PUSH` 套接字，则消息均匀分布在所有的 `PULL` 套接字中；在实践中，它们是负载均衡的（点对点负载平衡！）。另一方面，从多个 `PUSH` 套接字接收消息的 `PULL` 套接字将使用公平排队系统处理消息，这意味着它们将从所有负载是均衡的。
- 通过没有任何连接的 `PULL` 套接字的 `PUSH` 套接字发送的消息不会丢失；他们排队等待生产者，直到一个节点联机并开始提取消息。

我们现在开始了解 `ØMQ` 与传统 `Web` 服务的不同之处，它如何成为构建任何类型的消息传递系统的理想工具。

使用ØMQ构建分布式hashsum cracker

现在是时候构建一个示例应用程序来查看我们刚刚描述的 `PUSH / PULL` 套接字的属性。 一个简单而迷人的应用程序可能是一个 `hashsum cracker` ，一个使用暴力破解技术来尝试将给定的 `hashsum（MD5，SHA1等）` 与给定字母表中每个可能的字符变体进行匹配的系统。 这个算法的负载量是很高的（ http://en.wikipedia.org/wiki/Embarrassingly_parallel ），它非常适合构建演示并行管道功能的示例。

对于我们的应用程序，我们希望通过一个节点来实现典型的并行管道，以在多个 `worker` 之间创建和分配任务，以及一个节点来收集所有结果。我们刚刚描述的系统可以使用以下体系结构在 `ØMQ` 中实现：



在我们的体系结构中，我们有一个 `ventilator` ，用于生成给定字母表中所有可能的字符变体，并将它们分发给一组 `worker` ，然后计算每个给定变体的哈希函数并尝试将其与输入的哈希函数进行匹配。如果找到匹配项，则结果将发送到结果收集器节点（ `sink` ）。

重点是 `ventilator` 和 `sink` ，而 `worker` 节点是随时在变化中的。这意味着每个 `worker` 将其 `PULL` 套接字连接到 `ventilator` ，并将其 `PUSH` 套接字连接到 `ventilator` ；通过这种方式，我们可以在不改变 `ventilator` 和 `sink` 中的任何参数的情况下，启动和停止我们想要的 `worker` 数量。

实现**ventilator**

现在，让我们开始通过在名为 `ventilator.js` 的文件中为 `ventilator` 创建一个新模块来实现我们的系统：

```
const zmq = require('zmq');
const variationsStream = require('variations-stream');
const alphabet = 'abcdefghijklmnopqrstuvwxyz';
const batchSize = 10000;
const maxLength = process.argv[2];
const searchHash = process.argv[3];

const ventilator = zmq.socket('push');  // [1]
ventilator.bindSync("tcp://*:5016");

let batch = [];
variationsStream(alphabet, maxLength)
  .on('data', combination => {
    batch.push(combination);
    if (batch.length === batchSize) {  // [2]
      const msg = {searchHash: searchHash, variations: batch};
      ventilator.send(JSON.stringify(msg));
      batch = [];
    }
  })
  .on('end', () => {
    //send remaining combinations
    const msg = {searchHash: searchHash, variations: batch};
    ventilator.send(JSON.stringify(msg));
  })
;
```

为避免产生太多变化，我们的生成器只使用英文字母的小写字母，并对生成的单词的大小设置限制。这个限制在输入中作为命令行参数（ `maxLength` ）与 `hashsum` 匹配（ `searchHash` ）一起提供。 我们使用名为variation-stream的库来使用流式接口生成所有变体。

但是我们最感兴趣分析的部分是我们如何给 `worker` 分配任务：

1. 我们首先创建一个 `PUSH` 套接字，并将其绑定到本地端口 `5000` ；这是 `worker` 的 `PULL` 套接字将连接以接收任务的地方。
2. 我们将每个批次生成的变体进行分组，然后制作一条消息，其中包含匹配的散列和要检查的一批单词。这实质上是 `worker` 将接受的任务对象。当我们通过 `ventilator` 套接字调用 `send()` 时，消息将按循环分配传递给下一个可用的 `worker` 。

实现worker

现在是实现 `worker` （ `worker.js` ）的时候了：

```
const zmq = require('zmq');
const crypto = require('crypto');
const fromVentilator = zmq.socket('pull');
const toSink = zmq.socket('push');

fromVentilator.connect('tcp://localhost:5016');
toSink.connect('tcp://localhost:5017');

fromVentilator.on('message', buffer => {
  const msg = JSON.parse(buffer);
  const variations = msg.variations;
  variations.forEach( word => {
    console.log(`Processing: ${word}`);
    const shasum = crypto.createHash('sha1');
    shasum.update(word);
    const digest = shasum.digest('hex');
    if (digest === msg.searchHash) {
      console.log(`Found! => ${word}`);
      toSink.send(`Found! ${digest} => ${word}`);
    }
  });
});
```

正如我们所说的，我们的 `worker` 在我们的体系结构中代表了一个临时节点，因此，它的套接字应连接到远程节点，而不是侦听传入连接。这正是我们在 `worker` 中所做的，我们创建了两个套接字：

- 连接到 `ventilator` 的 `PULL` 套接字
- 用于接收任务连接到接收器的 `PUSH` 套接字，用于传播结果

除此之外，我们的 `worker` 完成的工作非常简单：对于收到的每条消息，我们迭代它包含的一批单词，然后对每个单词计算 `SHA1` 校验和，并尝试将其与针对消息传递的 `searchHash` 进行匹配。当找到匹配时，结果被转发到接收器。

实现**sink**

对于我们的例子来说，接收器是一个非常基本的结果收集器，它只是将 `worker` 接收的消息打印到控制台。 文件 `sink.js` 的内容如下所示：

```
const zmq  = require('zmq');
const sink = zmq.socket('pull');
sink.bindSync("tcp://*:5017");

sink.on('message', buffer => {
  console.log('Message from worker: ', buffer.toString());
});
```

运行应用

我们现在准备运行我们的应用程序；让我们开始几个 `worker` 和 `sink` ：

```
node worker
node worker
node sink
```

然后启动 `ventilator` ，指定要生成的单词的最大长度以及我们希望匹配的 `SHA1` 校验和。以下是参数的示例列表：

```
node ventilator 4 f8e966d1e207d02c44511a58dccff2f5429e9a3b
```

当运行上述命令时， `ventilator` 将开始生成所有可能的单词，其长度至多为四个字符，并将它们分配给我们开始的工作人员，以及我们提供的校验和。计算结果（如果有的话）将显示在接收器应用程序的终端中。

请求/回复模式

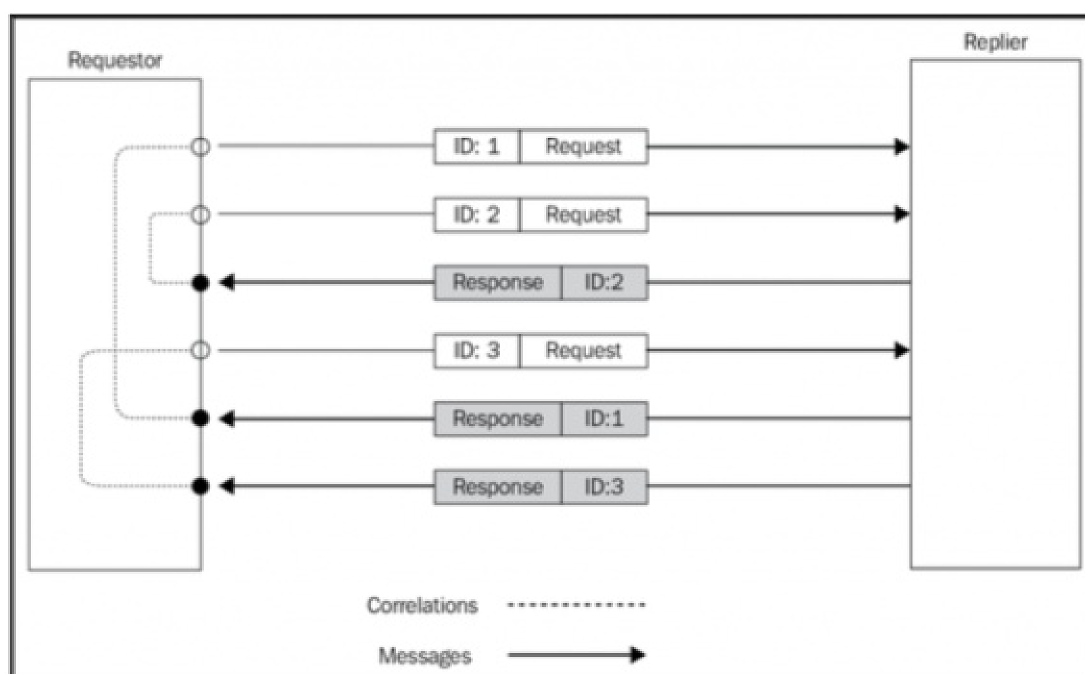
处理消息传递系统通常意味着使用单向异步通信;发布/订阅就是一个很好的例子。

单向通信可以在并行性和效率方面给我们带来巨大的优势，但单靠它们无法解决我们所有的集成和通信问题。有时候，一个很好的请求/回复模式可能只是这项工作的完美工具。因此，在所有那些我们拥有异步单向通道的情况下，知道如何构建一个允许我们以请求/回复方式交换消息的模式是很重要的。这正是我们接下来要学习的内容。

关联ID

我们将要学习的第一个请求/回复模式称为关联ID，它表示在单向通道之上构建请求/回复模式的基本内容。

该模式包括标记每个请求的标识符，然后由接收方附加到响应中；通过这种方式，请求的发送者可以关联这两个消息并将响应返回给正确的处理程序。这优雅地解决了存在单向异步通道的问题，消息可以随时在任何方向传播。我们来看看下图中的例子：



前面的场景显示了如何使用关联 `ID` 使我们能够将每个响应与正确的请求进行匹配，即使这些响应是以不同的顺序发送和接收的。

使用关联实现请求/答复模式

现在让我们开始通过选择最简单类型的单向通道（一个是点对点（它直接连接系统的两个节点）和一个全双工（消息可以双向传输））来进行尝试。

关于管道连接，我们可以找到例如 `WebSockets` ：它们在服务器和浏览器之间建立点对点连接，并且消息可以以任何方向传播。另一个例子是使用 `child_process.fork()` 生成子进程时创建的通信通道。我们应该已经知道了，我们在 `Chapter9-Advanced Asynchronous Recipes` 中看到了这个API。这个通道也是异步的：它只将父进程连接到子进程，并允许消息以任何方向传播。这可能是这个类别的最基本的渠道，所以这就是我们下一个例子中要用到的。

下一个应用程序的计划是构建一个抽象，以包装在父进程和子进程之间创建的通道隧道。这个抽象应该提供一个请求/回复通信隧道，通过用一个关联 `ID` 自动标记每个请求，然后将任何传入回复的 `ID` 与等待响应的请求处理程序列表进行匹配。

从 `Chapter9-Advanced Asynchronous Recipes` 中，我们应该记住父进程可以使用两个方法访问带有子进程的通道：

- `child.send(message)`
- `child.on('message',callback)`

以类似的方式，子进程可以使用以下方式访问父进程的通道：

- `process.send(message)`
- `process.on('message',callback)`

这意味着父进程中可用的隧道的 `API` 与子进程中可用的隧道的 `API` 相同；这将允许我们建立一个通用的方法，以便可以从通道的两端发送请求。

抽象request

我们通过考虑负责发送新请求的部分开始构建这个抽象请求；让我们创建一个名为 `request.js` 的新文件：

```
const uuid = require('node-uuid');

module.exports = channel => {
  const idToCallbackMap = {};  // [1]

  channel.on('message', message => {  // [2]
    const handler = idToCallbackMap[message.inReplyTo];
    if(handler) {
      handler(message.data);
    }
  });

  return function sendRequest(req, callback) {  // [3]
    const correlationId = uuid.v4();
    idToCallbackMap[correlationId] = callback;
    channel.send({
      type: 'request',
      data: req,
      id: correlationId
    });
  };
};
```

这就是我们的抽象请求的工作原理：

1. 看 `request()` 函数。该模式的神奇之处在于 `idToCallbackMap` 变量，它存储了传出请求与其回复处理程序之间的关联。
2. 一旦工厂被调用，我们所做的第一件事就是开始监听收到的消息。如果消息的关联 `ID` （包含在 `inReplyTo` 属性中）与 `idToCallbackMap` 变量中包含的任何 `ID` 相匹配，我们知道我们刚收到一个回复，因此我们获得了对相关响应处理程序的引用，并且用 消息中包含的数据。
3. 最后，我们返回我们将用来发送新请求的函数。 其工作是使用node-uuid生成关联 `ID` ，然后将请求数据包装起来，并指定关联ID `correlationId` 和消息类型 `type` 。

这就是 `request` 模块；让我们转到下一部分。

抽象reply

我们距实现完整的 `request/reply` 模式只有一步之遥，所以让我们看看 `request.js` 模块的对应的模块是如何工作的。我们创建另一个名为 `reply.js` 的文件，它将包含答复处理程序：

```
module.exports = channel =>
{
  return function registerHandler(handler) {
    channel.on('message', message => {
      if (message.type !== 'request') return;
      handler(message.data, reply => {
        channel.send({
          type: 'response',
          data: reply,
          inReplyTo: message.id
        });
      });
    });
  };
};
```

我们的 `reply` 模块又是一个工厂，它返回一个函数来注册新的答复处理程序。这是在注册新处理程序时发生的情况：

1. 我们开始监听传入的请求，当我们收到请求时，我们立即通过传递消息的数据和回调函数来收集处理程序的回复来调用处理程序。
2. `handler` 程序完成其工作后，它将调用我们提供的回调，并返回其答复。然后我们通过附加请求的关联 `ID` （ `inReplyTo` 属性）来构建，然后我们将所有内容都放回到隧道中。

关于这种模式的惊人之处在于，在 `Node.js` 中，它非常容易；我们所有的东西都是异步的，所以建立在单向通道之上的异步请求/回复通信与其他任何异步操作并没有太大的不同，特别是当我们构建一个抽象方法来隐藏其实现细节时。

尝试运行完整的**request/reply**模块

现在我们准备尝试运行我们新的异步request/reply模块。 让我们在一个名为 `replier.js` 的文件中创建一个示例 `replier` ：

```
const reply = require('./reply')(process);

reply((req, cb) => {
  setTimeout(() => {
    cb({sum: req.a + req.b});
  }, req.delay);
});
```

我们的 `replier` 只需计算两个接收到的数字之间的和，并在某个延迟（也在请求中指定）之后返回结果。这将允许我们验证响应的顺序也可能与我们发送请求的顺序不同，以确认我们的模块正在工作。

完成示例的最后一步是在名为 `requestor.js` 的文件中创建请求者，该文件还具有使用 `child_process.fork()` 启动 `replier` 的任务：

```
const replier = require('child_process')
                .fork(`${__dirname}/replier.js`);
const request = require('./request')(replier);

request({a: 1, b: 2, delay: 500}, res => {
  console.log('1 + 2 = ', res.sum);
  // 这应该是我们收到的最后一个回复，所以我们关闭了channel
  replier.disconnect();
});

request({a: 6, b: 1, delay: 100}, res => {
  console.log('6 + 1 = ', res.sum);
});
```

请求者启动 `replier` ，然后将其引用传递给我们的请求模块。然后，我们运行一些示例请求，并验证它们与收到的响应之间的关联是否正确。

要试用这个示例，只需启动 `requestor.js` 模块; 输出应该类似于以下内容：

```
6 + 1 = 7
1 + 2 = 3
```

```
..orrelation_id (zsh)
➜ 07_correlation_id git:(master) ✗ ls
README.txt    node_modules package.json replier.js    reply.js     request.js    requestor.js
➜ 07_correlation_id git:(master) ✗ node requestor.js
6 + 1 =  7
1 + 2 =  3
➜ 07_correlation_id git:(master) ✗
```

这证实了我们的模式完美地工作，并且 `reply` 与他们自己的请求正确地相关联，不管他们以什么顺序发送或接收。

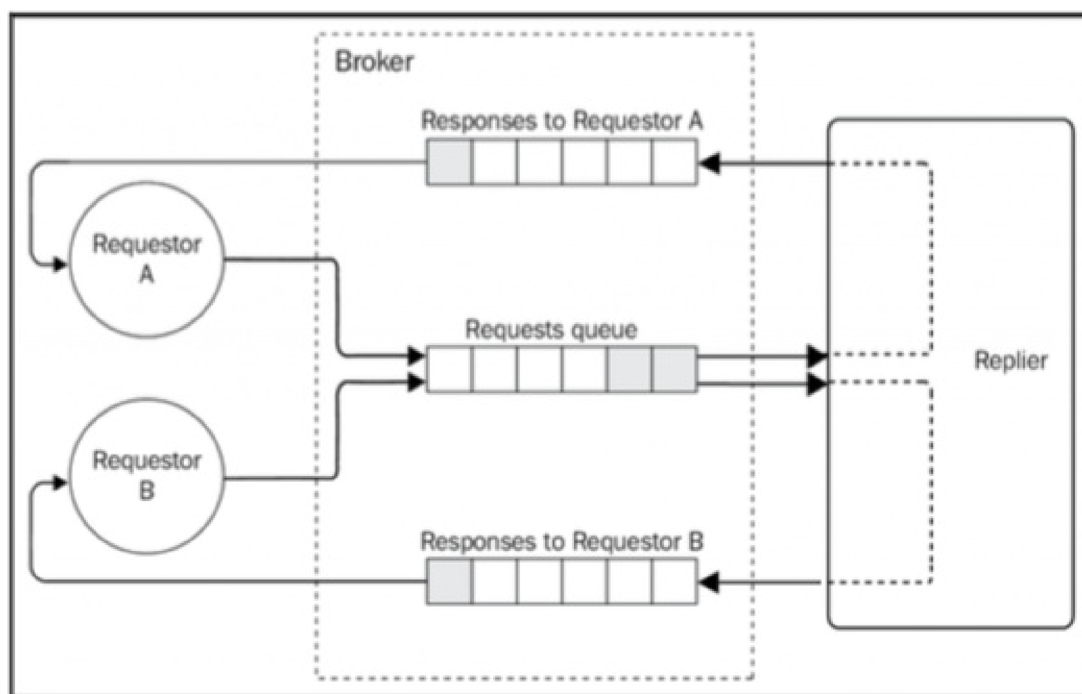
返回地址

关联 `ID` 是在单向信道之上创建请求/回复通信的基本模式；然而，当我们的消息架构拥有多个通道或队列，或者可能有多个请求者时，这还不够。在这些情况下，除了关联 `ID` 之外，我们还需要知道返回地址，这是允许回复者将回复发送回请求的原始发件人的一条信息。

在AMQP中实现返回地址模式

在 `AMQP` 中，返回地址是请求者正在侦听传入回复的队列。因为响应只能由一个请求者接收，所以队列是私有的并且不在不同的使用者之间共享是很重要的。从这些属性中，我们可以推断出我们将需要一个暂时队列，将其作用于请求者的连接，并且应答者必须与返回队列建立点对点通信，以便能够传递其响应。

以下为我们提供了这种情况的一个例子：



为了在 AMQP 上创建请求/应答模式，我们需要做的就是在消息属性中指定响应队列的名称；这样，回复者知道应答消息必须传送到哪里。 这个理论看起来非常简单，所以我们来看看如何在真正的应用程序中实现它。

实现request

现在让我们在 AMQP 之上构建一个请求/回复抽象。我们将使用 RabbitMQ 作为代理，但任何兼容的 AMQP 代理都应该可以完成这项工作。让我们从请求开始（在 amqpRequest.js 模块中实现）；我们只会在这里展示相关的部分。

第一件事情是我们如何创建队列来保存响应；看以下代码：

```
channel.assertQueue('', {exclusive: true});
```

当我们创建队列时，我们没有指定任何名字，这意味着我们会选择一个随机的名字；除此之外，队列是独占的，这意味着它被绑定到活动的 AMQP 连接，并且在连接关闭时它将被销毁。没有必要将队列绑定到交换机，因为我们不需要任何路由或分配到多个队列；这意味着消息必须直接传递到我们的响应队列中。

接下来，让我们看看我们如何产生一个新的请求：

```
class AMQPRequest {
  //...
  request(queue, message, callback) {
    const id = uuid.v4();
    this.idToCallbackMap[id] = callback;
    this.channel.sendToQueue(queue, new Buffer(JSON.stringify(me
ssage)), {
      correlationId: id,
      replyTo: this.replyQueue
    });
  }
}
```

`request()` 方法接受请求队列的名称和要发送的消息作为输入。正如我们在前一节中所了解的，我们需要生成一个关联 `ID` 并将其关联到回调函数。最后，我们发送消息，指定 `correlationId` 和 `replyTo` 属性作为元数据。

有趣的是，为了发送消息，我们使用 `channel.sentToQueue() API` 而不是 `channel.publish()` ；这是因为我们不希望使用交换机来实施任何发布/订阅分发，而是直接进入目标队列的更基本的点对点传递。

> 在 `AMQP` 中，我们可以指定一组要传递给消费者的属性（或元数据）以及主要消息。

我们的 `amqpRequest` 类的最后一个重要部分是我们监听传入响应的地方：

```
_listenForResponses() {
  return this.channel.consume(this.replyQueue, msg => {
    const correlationId = msg.properties.correlationId;
    const handler = this.idToCallbackMap[correlationId];
    if (handler) {
      handler(JSON.parse(msg.content.toString()));
    }
  }, {
    noAck: true
  });
}
```

在前面的代码中，我们监听我们明确创建的用于接收响应的队列中的消息，然后为每个传入消息读取关联 `ID` ，并将它与等待答复的处理程序列表进行匹配。一旦我们有了处理程序，我们只需要通过传递 `reply` 消息来调用它。

实现reply

这就是 `amqpRequest` 模块。现在是时候在名为 `amqpReply.js` 的新模块中实现响应对象。

在这里，我们必须保存传入请求的队列；我们可以为此使用一个简单的持久队列。我们不会展示这部分，因为它在所有 `AMQP` 都具有。我们感兴趣的是看到的是我们如何处理请求，然后将其发送回正确的队列：

```
class AMQPReply {
  //...
  handleRequest(handler) {
    return this.channel.consume(this.queue, msg => {
      const content = JSON.parse(msg.content.toString());
      handler(content, reply => {
        this.channel.sendToQueue(
          msg.properties.replyTo, // 这里保存的请求消息的队列
          new Buffer(JSON.stringify(reply)), {
            correlationId: msg.properties.correlationId
          }
        );
        this.channel.ack(msg);
      });
    });
  }
}
```

在发送 `reply` 时，我们使用 `channel.sendToQueue()` 将消息直接发布到消息的 `replyTo` 属性（我们的返回地址）中指定的队列中。我们的 `amqpReply` 对象的另一个重要任务是在回复对象中设置 `correlationId` ，以便接收者可以将消息与挂起的请求列表进行匹配。

实现requestor和replier

现在一切都准备好了，让我们首先尝试一下，但首先，让我们构建一个样本 `requestor` 和 `replier` ，从模块 `replier.js` 开始：

```
const Reply = require('./amqpReply');
const reply = Reply('requests_queue');

reply.initialize().then(() => {
  reply.handleRequest((req, cb) => {
    console.log('Request received', req);
    cb({sum: req.a + req.b});
  });
});
```

可以看到我们构建的模块如何处理关联 `ID` 和返回地址。我们所需要做的就是初始化一个新的 `reply` 对象，指定我们希望接收我们请求的队列的名称（ `requests_queue` ）。我们的样本重新计算接收到的两个数字的总和作为输入，并使用提供的回调函数返回结果。

另一方面，我们在 `requestor.js` 文件中实现了一个样例 `request` ：

```
const req = require('./amqpRequest')();

req.initialize().then(() => {
  for (let i = 100; i > 0; i--) {
    sendRandomRequest();
  }
});

function sendRandomRequest() {
  const a = Math.round(Math.random() * 100);
  const b = Math.round(Math.random() * 100);
  req.request('requests_queue', {a: a, b: b},
    res => {
      console.log(`${a} + ${b} = ${res.sum}`);
    }
  );
}
```

我们的示例请求程序将 `100` 个随机请求发送到 `requests_queue` 队列。在这种情况下，有趣的是我们完美地完成了它的工作，隐藏了异步请求/应答模式的所有细节。

现在，要尝试系统，只需运行 `replier` 程序模块和 `requestor` 模块：

```
node replier
node requestor
```

```
..orrelation_id (zsh)
➜ 07_correlation_id git:(master) ✗ ls
README.txt   node_modules package.json replier.js   reply.js    request.js   requestor.js
➜ 07_correlation_id git:(master) ✗ node requestor.js
6 + 1 =  7
1 + 2 =  3
➜ 07_correlation_id git:(master) ✗
```

我们会看到 `requestor` 发布的一系列操作，然后由 `replier` 收到，然后回复 `response` 。

现在我们可以尝试其他实验。一旦 `replier` 第一次启动，它会创建一个持久队列；这意味着，如果我们现在停止并再次运行请求者，则不会有任何请求丢失。 所有消息都将存储在队列中，直到重新启动重新启动。

这些都是因为我们使用了 `AMQP` 。 为了测试这个假设，我们可以尝试启动两个或更多的 `replier` 实例，并观察它们之间的负载平衡请求。这是有效的，因为每次 `requestor` 启动时，它将自己作为一个监听器附加到同一个持久队列中，结果，代理将负载均衡队列中所有消费者的消息同步到这里。

总结

我们已经到了本章的结尾。在这里，我们了解了最重要的消息传递和集成模式以及它们在分布式系统设计中扮演的角色。我们熟悉了三种主要类型的消息交换模式：发布/订阅，管道和请求/回复，并且我们看到了如何使用对等体系结构或消息代理来实现它们。我们分析了他们的优缺点，我们发现通过使用 `AMQP` 可以给我们提供更大的便捷，我们可以实现可靠和可扩展的应用程序，而只需很少的开发工作，但需要花费更多系统来维护和扩展我们应用程序。此外，我们看到了 `ØMQ` 如何让我们构建分布式系统，以便我们可以全面控制架构的每个方面，根据自己的需求对其属性进行微调。

本章是本书的最后一章，到现在为止，我们应该有一个基本概念，以及基本了解了 `Node.js` 可以用在我们的项目中应用的模式和技术。我们还应该更深入地了解 `Node.js` 的开发方式，以及它的优缺点。在整本书中，我们也有机会使用到很多别的开发人员开发的包和库和解决方案。最后，这是 `Node.js` 最漂亮的一个方面：它的人员，一个人人都可以在回馈某些东西时发挥作用的社区。

希望有一天你也可以给 `Node.js` 社区作出贡献。

Node.js-Design-Patterns

Welcome to the Node.js Platform

最新ES语法

reactor模式

`reactor模式` 是 `Node.js` 异步编程的核心模块，其核心概念是： `单线程` 、 `非阻塞I/O` 。

- `非阻塞I/O` ：在这种机制下，后续代码块不会等到 `I/O` 请求数据的返回之后再执行。如果当前时刻所有数据都不可用，函数会先返回预先定义的常量值(如 `undefined`)，表明当前时刻暂无数据可用。例如，在 `Unix` 操作系统中， `fcntl()` 函数操作一个已存在的文件描述符，改变其操作模式为 `非阻塞I/O` (通过 `O_NONBLOCK` 状态字)。一旦资源是非阻塞模式，如果读取文件操作没有可读取的数据,或者如果写文件操作被阻塞,读操作或写操作返回 `-1` 和 `EAGAIN` 错误。 `非阻塞I/O` 最基本的模式是通过轮询获取数据，这也叫做忙-等模型。看下面这个例子，通过 `非阻塞I/O` 和轮询机制获取 `I/O` 的结果。

事件多路复用

```
let socketA, pipeB;
wachedList.add(socketA, FOR_READ);
wachedList.add(pipeB, FOR_READ);
while(events = demultiplexer.watch(wachedList)) {
  // 事件循环
  foreach(event in events) {
    // 这里并不会阻塞，并且总会有返回值（不管是不是确切的值）
    data = event.resource.read();
    if (data === RESOURCE_CLOSED) {
      // 资源已经被释放，从观察者队列移除
      demultiplexer.unwatch(event.resource);
    } else {
      // 成功拿到资源，放入缓冲池
      consumeData(data);
    }
  }
}
```

Node.js Essential Patterns

回调模式

回调模式分为异步CPS风格和同步CPS风格、原生JS也可以实现回调模式

```
function add(a, b) {
  return a + b;
}
```

改为CPS风格：

```
function add(a, b, callback) {
  callback(a + b);
}
```

异步CPS：

```
function additionAsync(a, b, callback) {
 setTimeout(() => callback(a + b), 100);
}
```

Zalgo

解决方案：

- 使用同步API
- 延时处理

Node.js的惯用风格

- 错误处理总在最前
- 错误传播
- 某些异常不太好捕获

模块系统及其模式

- 模块缓存原理

```
const require = (moduleName) => {
  console.log(`Require invoked for module: ${moduleName}`);
  const id = require.resolve(moduleName);
  // 是否命中缓存
  if (require.cache[id]) {
    return require.cache[id].exports;
  }
  // 定义module
  const module = {
    exports: {},
    id: id
  };
  // 新模块引入，存入缓存
  require.cache[id] = module;
  // 加载模块
  loadModule(id, module, require);
  // 返回导出的变量
  return module.exports;
};
require.cache = {};
require.resolve = (moduleName) => {
  /* 通过模块名作为参数resolve一个完整的模块 */
};
```

- 模块循环依赖
- 模块寻找的算法

观察者模式

- EventEmitter类 如何让任意对象可观察，拓展EventEmitter类：

```
const EventEmitter = require('events').EventEmitter;
const fs = require('fs');
class FindPattern extends EventEmitter {
  constructor(regex) {
    super();
    this.regex = regex;
    this.files = [];
  }
  addFile(file) {
    this.files.push(file);
    return this;
  }
  find() {
    this.files.forEach(file => {
      fs.readFile(file, 'utf8', (err, content) => {
        if (err) {
          return this.emit('error', err);
        }
        this.emit('fileread', file);
        let match = null;
        if (match = content.match(this.regex)) {
          match.forEach(elem => this.emit('found', file, elem));
        }
      });
    });
    return this;
  }
}
```

Asynchronous Control Flow Patterns with Callbacks

如何写更优雅的回调：

- 避免回调地狱
- 迭代模式

```
function iterate(index) {
  if (index === tasks.length) {
    return finish();
  }
  const task = tasks[index];
  task(function() {
    iterate(index + 1);
  });
}

function finish() {
  // 迭代完成的操作
}

iterate(0);
```

- 并发处理

Asynchronous Control Flow Patterns with ES2015 and Beyond

这一章主要讲的是如何用 `Promise` 、 `Generator` ，以及 `async await` 简化异步。

几种方式各有优劣：

方案	Pros	Cons
扁平的 JavaScript（Plain JavaScript）	● 不需要任何库或技术 ● 提供最好的性能 ● 提供与第三方库最佳的兼容性 ● 允许 ad hoc 和更高级算法的创建	可能需要额外的代码和相对复杂的算法
Async（library）	● 简化最常见的控制流模式 ● 还是一个 callback-based 的解决方案 ● 性能好	● 引入一个外部依赖 ● 对于高级的流来说还是不够的
Promises	● 大大简化最常见的控制流模式 ● 鲁棒的 error 处理 ● ES2015 规范的一部分 ● 确保 onFullfilled 和 onRejected 的延迟调用	● 需要 promisify callback-based 的 APIs ● 引入以小的性能损失
Generators	● 使得非阻塞 API 看起来像阻塞一样 ● ES2015 规范的一部分	● 需要一个辅助的控制流库 ● 依然需要 callbacks 或 promises 来实现非顺序流 ● 需要 thunkify 或 promisify 非 generator-based 的 APIs
Async await	● 使得非阻塞 API 看起来像阻塞一样 ● 简洁直观的语法	● 在原生的 JavaScript 和 Node.js 还不能使用 ● 今天使用需要 Babel 或其他翻译器和一些配置

Coding with Streams

Streams和Buffer

- 空间效率更高
- 时间效率更高

实现可读的Streams

```
const stream = require('stream');
const Chance = require('chance');

const chance = new Chance();

class RandomStream extends stream.Readable {
  constructor(options) {
    super(options);
  }

  _read(size) {
    const chunk = chance.string(); //[1]
    console.log(`Pushing chunk of size: ${chunk.length}`);
    this.push(chunk, 'utf8'); //[2]
    if (chance.bool({
        likelihood: 5
      })) { //[3]
      this.push(null);
    }
  }
}

module.exports = RandomStream;
```

实现可写的Streams

```
const stream = require('stream');
const fs = require('fs');
const path = require('path');
const mkdirp = require('mkdirp');

class ToFileStream extends stream.Writable {
  constructor() {
    super({
      objectMode: true
    });
  }

  _write(chunk, encoding, callback) {
    mkdirp(path.dirname(chunk.path), err => {
      if (err) {
        return callback(err);
      }
      fs.writeFile(chunk.path, chunk.content, callback);
    });
  }
}
module.exports = ToFileStream;
```

双重的Streams

```
const stream = require('stream');
const util = require('util');

class ReplaceStream extends stream.Transform {
  constructor(searchString, replaceString) {
    super();
    this.searchString = searchString;
    this.replaceString = replaceString;
    this.tailPiece = '';
  }

  _transform(chunk, encoding, callback) {
    const pieces = (this.tailPiece + chunk)         //[1]
      .split(this.searchString);
    const lastPiece = pieces[pieces.length - 1];
    const tailPieceLen = this.searchString.length - 1;

    this.tailPiece = lastPiece.slice(-tailPieceLen);     //[2]
    pieces[pieces.length - 1] = lastPiece.slice(0,-tailPieceLen)
;

    this.push(pieces.join(this.replaceString));       //[3]
    callback();
  }

  _flush(callback) {
    this.push(this.tailPiece);
    callback();
  }
}

module.exports = ReplaceStream;
```

异步

Streams在异步编程有很广泛的运用，实现一个无序并行的Streams：

```
const stream = require('stream');

class ParallelStream extends stream.Transform {
  constructor(userTransform) {
    super({objectMode: true});
    this.userTransform = userTransform;
    this.running = 0;
    this.terminateCallback = null;
  }

  _transform(chunk, enc, done) {
    this.running++;
    this.userTransform(chunk, enc, this._onComplete.bind(this),
this.push.bind(this));
    done();
  }

  _flush(done) {
    if(this.running > 0) {
      this.terminateCallback = done;
    } else {
      done();
    }
  }

  _onComplete(err) {
    this.running--;
    if(err) {
      return this.emit('error', err);
    }
    if(this.running === 0) {
      this.terminateCallback && this.terminateCallback();
    }
  }
}

module.exports = ParallelStream;
```

实现组合的Streams

使用诸如 `multipipe` 之类的库，我们可以通过组合一些核心库中已有的 `Streams` （文件 `combinedStreams.js` ）来轻松地构建组合的 `Streams` ：

```
const zlib = require('zlib');
const crypto = require('crypto');
const combine = require('multipipe');
module.exports.compressAndEncrypt = password => {
  return combine(
    zlib.createGzip(),
    crypto.createCipher('aes192', password)
  );
};
module.exports.decryptAndDecompress = password => {
  return combine(
    crypto.createDecipher('aes192', password),
    zlib.createGunzip()
  );
};
```

Design Patterns

- 工厂模式（`Factory`）：通过stampit可以实现组合的工厂函数，在Node.js中这种模式有广泛应用，例如Node.js的核心库http，也是提供了工厂创建实例的方式。
- 揭示构造模式（`Revealing constructor`）：揭示构造函数模式接受执行函数executor作为参数，这个函数被提供给构造函数，并在内部调用。这种模式也在Node.js有广泛应用。最为显著的是原生Promise使用了这一种模式。也可以通过揭示构造函数模式创建只读的event emitter，可以较好地保证函数内部安全性。
- 代理模式（`Proxy`）、装饰者模式（`Decorator`）都常常使用对象增强和对象组合的方式书写。其各有优缺点，对象增强会改变主体对象，对象组合的写法又比较繁琐。他们都有广泛的应用，例如著名的Mongoose大量使用代理模式。AOP编程方式也是代理模式的应用。hooks这个库则是AOP的完美体现。装饰者模式的应用也很多，如levelup的许多插件则使用了装饰者模式。而由于JavaScript语言的动态性，实现装饰者模式则比较简单。
- 适配器模式（`Adapter`）：允许我们用不同的接口去访问对象的功能，它适配一个对象，以便于它可以被不同接口调用。适配器模式也有所应用场景，例如我们可以对核心库做上层封装，并且适配对应的核心库的功能，书上的fsAdapter则是这个模式的较好的体现。
- 策略模式（`Strategy`）、状态模式（`State`）和模板模式（`Template`）：使用模式来简化大量的条件代码的书写，状态模式类似于策略模式，状态模式Context的策略会根据State的变化而变化。而模板模式其实就是C++的类模板在JavaScript的体现。由于JavaScript语言本身没有类模板这样的功能。通过在一个类的方法中抛出异常来实现一个抽象类和虚函数。
- 中间件模式（`Middleware`）：思想源于拦截过滤器模式和责任链模式。常见的Web框架Express和Koa都广泛使用中间件模式。

- 命令模式（ `Command` ）：降低了对象之间的耦合度，设计命令也相对简单，代码解耦，但是使用命令模式可能导致系统命令类过多，这是命令模式的一大缺陷。

Writing Modules

如何去定义一个模块

- 一个模块应该具有可读性和可理解性，因为它应该专注于一件事
- 一个模块被表示为一个单独的文件，使得其更容易被识别
- 模块可以更容易地在不同的应用程序中复用

依赖注入

依赖注入（DI）模式可能是软件设计中最容易被误解的概念之一。许多人将这个术语与框架和依赖注入容器相关联，例如 `Spring` （用于 `Java` 和 `C#` ）或 `Pimple` （用于 `PHP` ），但实际上它是一个很简单的概念。依赖注入模式背后的主要思想是由外部实体提供输入的组件的依赖关系。

这样的实体可以是客户端组件或全局容器，它集中了系统所有模块的关联。这种方法的主要优点是解耦，特别是对于取决于有状态实例的模块。使用DI，从外部接收每个依赖项，而不是硬编码到模块中。这意味着模块可以配置为其中的依赖关系，因此可以在不同的上下文中重用。

服务定位器

服务定位器核心原则是拥有一个中央注册中心，以便管理系统组件，并在模块需要加载依赖时作为中介。这个想法是要求服务定位器所连接的是依赖注入模块，而不是硬编码模块。通过使用服务定位器，我们引入了对它的依赖关系，它连接到模块的方式决定了它们的耦合程度，其可重用性较高。 在 `Node.js` 中，我们可以确定三种类型的服务定位器，区分它们的关键因素是它们连接到系统各个组件的方式：分为硬编码依赖服务定位器、依赖注入服务定位器和全局注入服务定位器。

服务定位器的基本模式：

```
"use strict";

module.exports = () => {
  const dependencies = {};
  const factories = {};
  const serviceLocator = {};

  serviceLocator.factory = (name, factory) => {
    factories[name] = factory;
  };

  serviceLocator.register = (name, instance) => {
    dependencies[name] = instance;
  };

  serviceLocator.get = (name) => {
    if (!dependencies[name]) {
      const factory = factories[name];
      dependencies[name] = factory && factory(serviceLocator);
      if (!dependencies[name]) {
        throw new Error('Cannot find module: ' + name);
      }
    }
    return dependencies[name];
  };

  return serviceLocator;
};
```

Advanced Asynchronous Recipes

这一章主要讲了三种书写异步的常见模型：

- 异步引入模块并初始化
- 在高并发的应用程序中使用批处理和缓存异步操作的性能优化
- 运行与 `Node.js` 处理并发请求的能力相悖的阻塞事件循环的同步 `CPU` 绑定操作

异步初始化模块

如果一个Node.js模块需要异步初始化，可以有以下两种解决方案：一是在开始使用模块之前之前确保模块已经初始化，否则则等待其初始化。二是使用预初始化队列进行初始化，在初始化之前的所有操作均放入预初始化队列中，等待初始化完成后取出队列中的任务。

- 等待初始化

```
// 模块app.js
const db = require('aDb'); // aDb是一个异步模块
const findAllFactory = require('./findAll');
db.on('connected', function() {
  const findAll = findAllFactory(db);
  // 之后再执行异步操作
});


// 模块findAll.js
module.exports = db => {
  //db 在这里被初始化
  return function findAll(type, callback) {
    db.findAll(type, callback);
  }
}
```

- 预初始化队列

```
const asyncModule = module.exports;

asyncModule.initialized = false;
asyncModule.initialize = callback => {
  setTimeout(() => {
    asyncModule.initialized = true;
    callback();
  }, 10000);
};

asyncModule.tellMeSomething = callback => {
  process.nextTick(() => {
    if(!asyncModule.initialized) {
      return callback(
        new Error('I don\'t have anything to say right now')
      );
    }
    callback(null, 'Current time is: ' + new Date());
  });
};
```

批处理和缓存

对于接口请求量较大的API，我们可以使用批处理或者缓存来提升接口性能：批处理指的是在调用异步函数的同时在队列中还有另一个尚未处理的回调，我们可以将回调附加到已经运行的操作上，而不是创建一个全新的请求。缓存不只可以是内存中的变量，还可以是数据库，甚至是专门的缓存服务器。此外，使用缓存需要有一定的策略对缓存进行淘汰，例如LRU。

CPU-bound任务

对于运算量较大的同步的CPU-bound任务，可能造成接口的阻塞。解决方案有两种：一种是使用异步来包装同步的CPU-bound任务，二是使用多进程，一般来说使用Node.js子进程，因为Node.js自带子进程模块，并且可以使用相关接口进行父子进程的管道通信，而如果子进程不是Node.js进程，一般只能通过标准输入输入来进行父子进程的通信。

如何父子进程进行通信：

```
const EventEmitter = require('events').EventEmitter;
const ProcessPool = require('./processPool');
const workers = new ProcessPool(__dirname + '/subsetSumWorker.js'
, 2);

class SubsetSumFork extends EventEmitter {
  constructor(sum, set) {
    super();
    this.sum = sum;
    this.set = set;
  }

  start() {
    workers.acquire((err, worker) => {  // [1]
      worker.send({sum: this.sum, set: this.set});

      const onMessage = msg => {
        if (msg.event === 'end') {  // [3]
          worker.removeListener('message', onMessage);
          workers.release(worker);
        }

        this.emit(msg.event, msg.data);  // [4]
      };

      worker.on('message', onMessage);  // [2]
    });
  }
}

module.exports = SubsetSumFork;
```

Scalability and Architectural Patterns

`cluster` 模块

`cluster` 模块主进程负责产生大量进程（ `worker` ），每个进程代表我们想要扩展的应用程序的一个实例。每个传入连接然后分布在克隆的 `worker` ，分散在他们的负载。

```
const cluster = require('cluster');
const os = require('os');

if(cluster.isMaster) {
  const cpus = os.cpus().length;
  for (let i = 0; i < cpus; i++) {  // [1]
    cluster.fork();
  }
} else {
  require('./app');  // [2]
}
```

零宕机重启

当代码需要更新时， `Node.js` 应用程序也可能需要重新启动。因此，在这种情况下，拥有多个实例可以帮助维护我们应用程序的可用性。 当我们不得不故意重新启动一个应用程序来更新它时，会出现一个小窗口，在这个窗口中应用程序将重新启动并且无法为请求提供服务。如果我们正在更新我们的个人博客，这是可以接受的，但对于具有服务水平协议（ `SLA` ）的专业应用程序就不行了，或者作为持续交付过程的一部分经常更新的专业应用程序。解决方案是实现零宕机重新启动，更新应用程序的代码而不影响其可用性。

使用 `cluster` 模块，这又是一项非常简单的任务；该模式包括一次重启一个 `worker` 。这样，剩余的 `worker` 可以继续操作和维护可用应用程序的服务。

然后，让我们将这个新模块添加到我们的集群服务器；我们所要做的就是添加一些由主进程执行的新代码（看 `clusteredApp.js` 文件）：

```
const cluster = require('cluster');
const os = require('os');

if (cluster.isMaster) {
  const cpus = os.cpus().length;
  for (let i = 0; i < cpus; i++) {
    cluster.fork();
  }

  cluster.on('exit', (worker, code) => {
    if (code != 0 && !worker.exitedAfterDisconnect) {
      console.log('Worker crashed. Starting a new worker');
      cluster.fork();
    }
  });

  process.on('SIGUSR2', () => {
    console.log('Restarting workers');
    const workers = Object.keys(cluster.workers);

    function restartWorker(i) {
      if (i >= workers.length) return;
      const worker = cluster.workers[workers[i]];
      console.log(`Stopping worker: ${worker.process.pid}`);
      worker.disconnect();

      worker.on('exit', () => {
        if (!worker.suicide) return;
        const newWorker = cluster.fork();
        newWorker.on('listening', () => {
          restartWorker(i + 1);
        });
      });
    }
    restartWorker(0);
  });
} else {
  require('./app');
}
```

粘性负载均衡

- 反向代理
- nginx的负载均衡

Messaging and Integration Patterns

常见三类消息传递方式：

- 发布/订阅模式
- 管道和任务分配模式
- 请求/回复模式

三类模式都可以点对点通信或者是使用代理进行通信。